НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«До захисту допущено»

Завідувач кафедри
_____ Романкевич В.О.
(підпис) (ініціали, прізвище)

“___”_____2020 р.

Дипломний проект

на здобуття ступеня бакалавра

зі спеціальності: **123 Комп‘ютерна інженерія**

на тему:«Комплекс программ для визначення нероздільних завадостійких кодів»

Виконав : студент VI курсу, групи КВ-63

Кравчук Володимир Вікторович _____
(підпис)

Керівник: доц. каф. СПіСКС,к. т. н., с.н.с. Тесленко О.К. _____
(підпис)

Консультант з нормоконтролю: к. т. н., доцент Клятченко Я.М. _____
(підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»**

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 123 «Комп‘ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Романкевич В.О.
(підпис) (ініціали, прізвище)
«___»_____2020р.

ЗАВДАННЯ
на дипломний проєкт студента
Кравчука Володимира Вікторовича

1.Тема проекту«Комплекс програм для визначення нероздільних завадостійких кодів»,
керівник проекту доц. каф. СПіСКС,к. т. н., с.н.с. Тесленко О.К.,
затверджені наказом по університету від «__» _____ 2020 р. №_____

2. Термін подання студентом проекту _____

3. Вихідні дані до проекту: технічна література на тему теорії завадостійкого кодування та алгоритму Брона-Кербоша, документація мови програмування Java та додаткових бібліотек.

4. Зміст пояснювальної записки:

Теоретичні відомості про завадостійке кодування та алгоритм Брона-Кербоша.

Удосконалення алгоритму для вирішення конкретної задачі пошуку завадостійких кодів.

Розробка та тестування програмного продукту. Опис його функціональності.

5. Перелік графічного матеріалу (із зазначенням обов’язкових креслеників, плакатів, презентацій тощо)

Д1. Покращений алгоритм Брона-Кербоша. Схема алгоритму.

Д2. Алгоритм побудови графа Хемінга. Схема алгоритму.

Д3. Алгоритм визначення мінімальної кодової відстані. Схема алгоритму.

Д4. Комплекс для визначення завадостійких кодів. Схема структурна.

6. Консультанти розділів проекту*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
нормоконтроль	к. т. н., доцент Клятченко Я.М.		

7. Дата видачі завдання_____

Календарний план

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1	Видача завдання на дипломне проектування	22.03.2020	
2	Вивчення літератури за тематикою роботи	05.04.2020	
3	Вибір засобів розробки	07.04.2020	
4	Реалізація вибраних алгоритмів	25.04.2020	
5	Розробка графічного інтерфейсу	03.05.2020	
6	Тестування програмного комплексу	05.05.2020	
7	Підготовка пояснювальної записки	10.05.2020	
8	Підготовка графічних матеріалів	18.05.2020	

Студент _____ _____
(підпис) (ініціали, прізвище)

Керівник проекту _____ _____
(підпис) (ініціали, прізвище)

Анотація

Бакалаврський проєкт включає пояснювальну записку (55 с., 45 рис., 4 додатки).

*Консультантом не може бути зазначено керівника дипломного проекту.

В даній роботі досліджена тема завадостійкого кодування та пошуку максимальної кліки на графі. Розглянуто різні типи кодування, описана проблема аналітичної швидкості коду, проаналізовано алгоритм Брона-Кербоша для пошуку клік. На основі особливостей еквівалентних кодів та графа Хемінга, запропоновано способи покращення алгоритму для вирішення задачі пошуку максимального нероздільного завадостійкого коду. Було вирішено розробити комплекс програм, який допоможе спростити визначення та дослідження нероздільних завадостійких кодів.

Було сформовано конкретні вимоги та функціональність для комплексу, а саме: можливість пошуку максимальних нероздільних завадостійких кодів відповідно до заданих користувачем параметрів, зупинка роботи комплексу в певний момент часу із збереженням проміжних даних з якими працював алгоритм, завантаження збережених даних та продовження роботи після зупинки, можливість виконання різних операцій над кодами, таких як визначення мінімальної кодової відстані, визначення кодової відстані кодослова до коду, сортування коду, надання користувачу простого та зрозумілого графічного інтерфейсу для зручності роботи з програмою.

Комплекс програм реалізований мовою програмування Java, яка підтримується усіма популярними операційними системами, з використанням стандартної бібліотеки JavaFX, для розробки графічних інтерфейсів.

Ключові слова: комплекс програм для визначення нероздільних завадостійких кодів, пошук максимальних кодів, алгоритм Брона-Кербоша, граф Хемінга, кодова відстань, відстань Хемінга, мова програмування Java, фреймворк JavaFX.

Abstract

The bachelors project includes an explanatory note (97 pages, 41 drawings, 7 annexes).

In this work, the topics of error correction and error detection coding, finding maximal clique of graph have been researched. Different types of coding were considered, the problem of analytic speed of code was described and Bron-Kerbosh algorithm was analyzed. Based on specifics of equivalents codes and Hamming graph the methods of algorithm optimization for finding maximal undivided error correcting code were suggested. It has been decided to develop a complex of program which will help to calculate and research error correcting codes.

The concrete requirements and functionality for the complex were formulated: possibility to search maximal undivided error correcting code according to parameters provided by user, stop work of complex in the moment with saving intermediate data algorithm are working with, loading the saved data and continue work after algorithm had been stopped, the possibility to perform some operations with codes like compute the minimal code distance, compute minimal code distance between a word and a code, sort code, provide simple and understandable graphical user interface for comfortable working with program.

The complex of programs is implemented by Java programming language which is supported by all the most popular operation systems using native library JavaFX for developing graphical user interface.

Key words: complex of programs for finding undivided error correcting codes, search of maximal codes, Bron-Kerbosh algorithm, Hamming Graph, code distance, Hamming distance, Java programming language, framework JavaFX.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.002	Комплекс програм для	4		
			визначення нероздільних			
			завадостійких кодів.			

			Технічне завдання			
	А4	ІАЛЦ.045490.003	Комплекс програм для	2		
			визначення нероздільних			
			завадостійких кодів.			
			Відомість технічного			
			проєкту			
	А4	ІАЛЦ.045490.004	Комплекс програм для	5		
			визначення нероздільних			
			завадостійких кодів.			
			Пояснювальна записка			
	А4	ІАЛЦ.045490.005	Покращений алгоритм	1		
			Брона-Кербоша.			
			Схема алгоритму			

Змін.	Арк.	№ докум.	Підпис	Дата		
Розробив		Кравчук В.В.				
Перевірив		Тесленко О.К.				
Консульт.						
Н. контроль		Клятченко Я.М.				
Зав. каф.		Романкевич В.О.				

ІАЛЦ.045490.001 ОА

Літ.	Аркуш	Аркушів
	1	2

Комплекс для визначення нероздільних завадостійких кодів

Опис альбому

КПІ ім. Ігоря Сікорського, ФПМ

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.006	Алгоритм побудови	1		

			графа Хемінга.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.007	Алгоритм визначення	1		
			мінімальної кодової			
			відстані.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.008	Комплекс для	1		
			визначення			
			завадостійких кодів.			
			Схема структурна.			
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

Зміст

					ІАЛЦ.045490.002 ТЗ		
Зм.	Арк.	№ докум.	Підп.	Дата			
Розроб.		Кравчук В.В.			Комплекс програм для визначення нероздільних завадостійких кодів	Літ.	Аркуш 1 / Аркушів 4
Перевір.		Тесленко О.К.					
Н.контр.		Клятченко Я.М.			Технічне завдання	КПІ ім. Ігоря Сікорського, ФПМ, КВ-63	
Затв.		Романкевич В.О.					

НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування роботи – дипломний проект на тему «Комплекс програм для визначення завадостійких кодів».

Область дослідження: спрощення визначення та дослідження нероздільних завадостійких кодів.

ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою даної роботи є дослідження методів завадостійкого кодування, аналіз нероздільних завадостійких кодів та алгоритму Брона-Кербоша. Визначення способів оптимізації алгоритму відповідно до конкретної задачі пошуку кодів на графі Хемінга. Реалізація програмного продукту.

ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації в періодичних виданнях та електронні статті у мережі Інтернет.

ТЕХНІЧНІ ВИМОГИ

Вимоги до до програмного продукту, що розробляється:

1. можливість пошуку максимальних нероздільних завадостійких кодів;
2. можливість зупинки роботи зі збереження проміжних результатів;
3. можливість продовження роботи на основі збережених результатів;
4. визначення мінімальної кодової відстані в коді;
5. визначення мінімальної кодової відстані кодослова до коду;
6. сортування коду;
7. наявність графічного інтерфейсу користувача;

Вимоги до апаратного забезпечення:

- Оперативна пам’ять: 4 Гб;

Вимоги до програмного та апаратного забезпечення користувача:

- Операційна система, яка підтримує JVM(Java Virtual Machine).
- Збірник проектів gradle.

ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1	Видача завдання на дипломне проектування	22.03.2020	
2	Вивчення літератури за тематикою роботи	05.04.2020	
3	Вибір засобів розробки	07.04.2020	
4	Реалізація вибраних алгоритмів	25.04.2020	
5	Розробка графічного інтерфейсу	03.05.2020	
6	Тестування програмного комплексу	05.05.2020	
7	Підготовка пояснювальної записки	10.05.2020	
8	Підготовка графічних матеріалів	18.05.2020	

Поз	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045492.004	Комплекс програм для	5		
			визначення нероздільних			
			завадостійких кодів.			
			Пояснювальна записка			
	А4	ІАЛЦ.045490.005	Покращений алгоритм	1		
			Брона-Кербоша.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.006	Алгоритм побудови	1		
			графа Хемінга.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.007	Алгоритм визначення	1		
			мінімальної кодової			
			відстані.			
			Схема алгоритму			
	А4	ІАЛЦ.045490.008	Комплекс для	1		
			визначення			
			завадостійких кодів.			
			Схема структурна.			

Зм.	Ар	№ докум.	Під	Да

ІАЛЦ.045490.003 ТП

Розробив	Кравчук В.В.	Комплекс програм для визначення нероздільних завадостійких кодів **Відомість технічного проєкту**	Літ. / Аркуш 1 / Аркушів 2
Перевірив	Тесленко О.К.		
Консульт.			КПІ ім. Ігоря Сікорського, ФПМ КВ-63
Н.контроль	Клятченко Я.М.		
Зав. каф.	Романкевич В.О.		

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

	Ар	№ докум.	Підпис	Да	ІАЛЦ.045490.003 ТП	А 2

ЗМІСТ

стор.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

V^n – множина всіх n-розрядних двійкових чисел;

Кодослово – елемент множини V^n ;

Відстань Хемінга між двома двійковими числами(кодословами) – кількість різних двійкових розрядів(кількість одиниць після порозрядної операції XOR)

XOR(виключне або) – логічна операція результатом якої є істина лише тоді коли операнди відрізняються.

Код C(t,n,d) – підмножина із V^n, з кількістю елементів t та відстанню Хемінга між будь якими кодословами не меншою ніж d.

Граф Хемінга – неорієнтовний граф, вершини якого належать V^n, а ребра між вершинами існують при умові, що відстань Хемінга між вершинами не менша за задану(*d*).

Кліка Графа Хемінга – сукупність вершин, де між будь якими вершинами із кліки існує ребро.

Повна кліка(повний код) – кліка, до якої не можливо додати ні одного кодослова.

Максимальна кліка(максимальний код) – кліка з найбільшою кількістю кодослів для даного графа Хемінга.

Впорядкований код – код, у якому кодослова розташовані у порядку зростання значень двійкових чисел.

Відстань кодослова до коду – найменша відстань Хемінга заданого кодослова до заданого коду.

Відстань між кодами – найменша відстань між будь яким кодословом із першого коду та будь-яким кодословом із другого коду.

IDE(*Integrated development environment*) - Інтегроване середовище розробки.

ПЗ – програмне забезпечення.

ВСТУП

У наш час галузь розробки програмного забезпечення дуже швидко розвивається, також з нею розвиваються мережеві та хмарні технології, не кажучи вже про Інтернет та телебачення. Всі ці системи повинні обмінюватися інформацією, між частинами однієї системи або одні системи з іншими. Передача інформації є невід'ємною частиною роботи будь-якої системи. Разом із цим з'являється потреба збереження цілісності інформації що передається. Якість передачі даних для кожної системи визначається окремо. Також покращуються канали передачі даних, проте повністю позбутися завад практично не можливо.

Для зменшення впливу завад на процес передачі даних і були створені завадостійкі коди. Вони дозволяють виявляти та виправляти помилки що виникли внаслідок впливу завад. Тому цій проблемі, яка є актуальною і в наш час, приділяють значної уваги дослідники та інженери. Було зроблено багато відкриттів та досліджень в сфері завадостійкого кодування. Знайдено достатньо блокових роздільних завадостійких кодів, але дослідження нероздільних кодів не такі значні.

В даній роботі було вирішено створити комплекс програм, який дозволить спростити визначення та дослідження нероздільних завадостійких кодів. В основі розробки лежать особливості графу Хемінга та алгоритм Брона-Кербоша. А для реалізації використано одну з найпопулярніших об'єктно орієнтованих мов програмування - Java.

1. ЗАВАДОСТІЙКЕ КОДУВАННЯ ТА АЛГОРИТМ БРОНА-КЕРБОША

1.2 Завадостійке кодування

Завадостійке кодування – це кодування, призначене для передачі даних по каналах із завадами, яке забезпечує виправлення або виявлення помилок утворених під час передачі. По суті, кодування – це додавання до вхідної інформації додаткової, перевірочної, інформації. А кількість цієї додаткової інформації – надлишковість, яка визначається за формулою:

$$k/(i+k) \quad , \tag{1}$$

де k – кількість додаткових біт; і – кількість інформаційних біт;

Для реалізації завадостійкого кодування використовують завадостійкі коди, яких на даний час уже є велика різноманітність (рис 1.1).

Тут виділяють два основні класи кодів : блокові та неперервні. Під час блокового кодування, кожному повідомленню відповідає блок (кодова комбінація) із сигналів. Кодуються і декодуються блоки окремо. В неперервних кодах введення керуючих сигналі у вхідний інформаційний потік відбувається без розбиття його на блоки. Процеси кодування та декодування відбуваються неперервно. Блокові коди класифікують на рівномірні, коли всі блоки мають однакову довжину та нерівномірні, якщо ця довжина змінюється. Рівномірні коди, у свою чергу, поділяють на роздільні, в яких існує чітке розмежування перевірочних та інформаційних символів, тоді як у нероздільних кодах такого розмежування немає. Також роздільні коди поділяють на лінійні та нелінійні. Більшість відомих лінійних кодів складають систематичні, у яких виконання лінійних операцій над певними інформаційними символами визначає значення перевірочних символів.

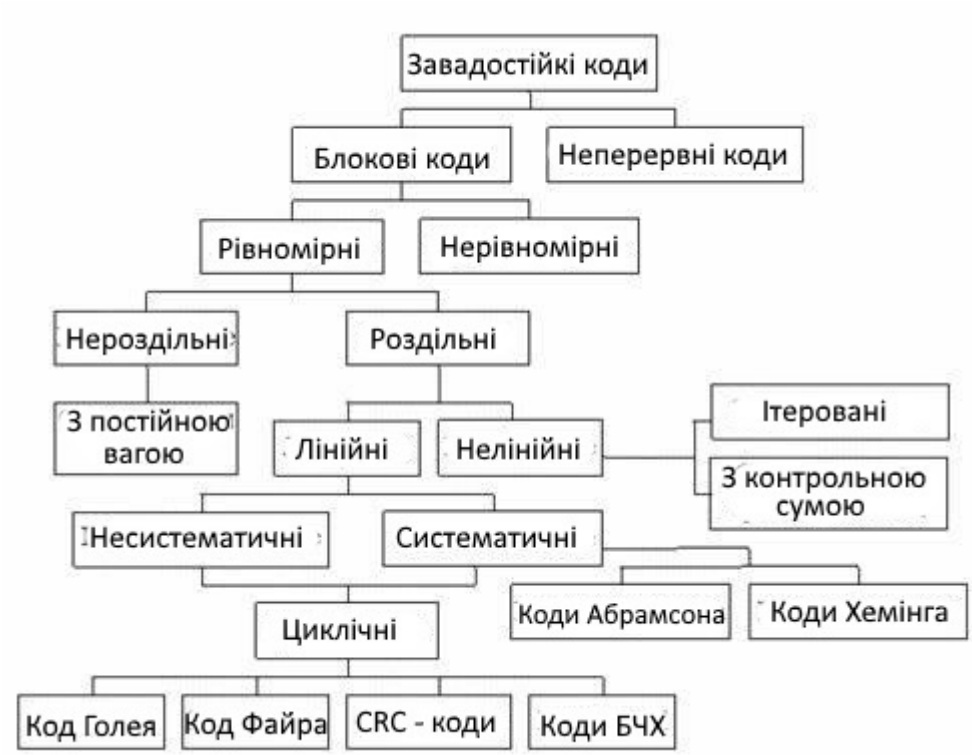


Рис. 1.1 Класифікація завадостійких кодів.

Одним із важливих параметрів коду є кодова відстань, для двійкових кодів – це мінімальна відстань Хемінга між двома кодословами.

Код C виправляє e або менше помилок, якщо кодова відстань d не менша ніж $2e+1$, тобто:

$$d \geq 2e+1 \qquad (2)$$

Для більшої наочності зображено сфери з радіусом e навколо кодослів у просторі заповненому наборами двійкових даних (рис 1.2). Для того, щоб кожна помилка кратності e була виправлена, слово з такою помилкою повинно знаходитись всередині лише однієї сфери, а для цього потрібно щоб сфери не перетинались. Таким чином, відстань між центрами двох сфер (кодословами) повинна задовольняти формулу (2).

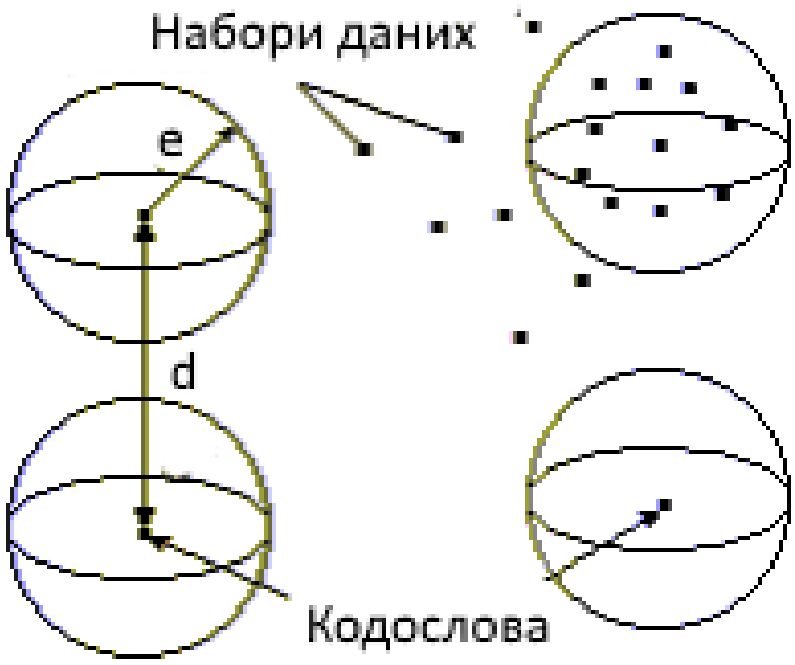


Рис. 1.2 Сфери навколо кодослів.

Щоб код *C* виявляв *e* або менше помилок, слово з такою помилкою не повинно бути кодовим, тоді кодова відстань буде:

$$d \geq e+1 \quad , \tag{3}$$

де *e* – кількість розрядів з помилкою або кратність помилки;

У роздільних кодах, щоб забезпечити виконання нерівності (2) до *m* інформаційних символів додають *k* додаткових. Яскравий приклад, це код Хемінга, який дозволяє виправляти одиночну(з кратністю 1) помилку. В коді Хемінга контрольні біти вставляються на позиціях кратних степеню числа 2 (1, 2, 4 …). Кожен такий біт відповідає за певний діапазон розрядів кодослова. Для такого коду, *k* має задовольняти нерівність:

$$2^k \geq k+m-1 \tag{4}$$

Код БЧХ(Боуза — Чоудхурі — Хоквінгема) – це шикорий клас циклічних кодів. Циклічні коди – це такі коди у яких кожна циклічна

перестановка кодослова, також є кодословом. Такі коди можуть виправляти помилки різної кратності. Для побудови таких кодів потрібно визначити довжину кодолова *n* та кодову відстань *d*:

$$n=2^h-1, \quad (5)$$

$$d=2^{h-1}-1, \quad (6)$$

де *h* – ціле, натуральне число. Таким чином, величина *n* може бути рівна 3, 7, 15 і т.д. Коди БЧХ мають непарне значення мінімальної кодової відстані[1]. А кількість інформаційних символів *m* та додаткових *k* визначаються із формул:

$$m=n-k \quad (7)$$

$$k=h(d-1)/2 \quad (8)$$

Досить широке застосування цих кодів зумовлене простотою реалізації кодерів та декодерів для них.

Проте, досягнення в області побудови блокових нероздільних кодів не такі значні. Під час нероздільного кодування групі інформаційних символів ставиться у відповідність нова группа кодуючих символів, які будуть передаватись по каналу, та не міститимуть у собі інформаційних символів у явному вигляді. Прикладому тут виступають коди Адамара. Ці коди своєю появою завдячують матриці Адамара (рис 1.3).

Матриця Адамара – *H* порядку *n* являє собою квадратну (*n*n)* матрицю із елементів +1 та -1, таку, що

$$H H^T=n E \ , \quad (9)$$

де *E* – одинична матриця[2].

Якщо в такій матриці замінити всі +1 на 0, а -1 на 1, то в отриманій матриці множина векторів-рядків утворить код, із відстанню Хемінга *n/2* між кодословами. Таким чином із будь якої матриці Адамара можна отримати код *C(n, n-1, n/2)*. Довжина коду *n-1* тому, що перший стовпчик матриці, у якому всі нулі, можна відкинути не змінюючи кодової відстані.

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$n=2$

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix},$$

$n=4$

Рис. 1.3 Приклади матриць Адамара.

За допомогою перетворень матриці можна побудувати ще два коди Адамара. Такі коди мають велику кодову відстань і можуть виправляти багато помилок але за це треба платити високою надлишковістю. Також, побудова матриць Адамара є не простою задачею, якій навіть присвячений окремий розділ сучасної комбінаторики[3].

Для практичного застосування завадостійких кодів потрібні кодери, пристрої які загодовують інформацію потрібним кодом, та декодери, які виконують зворотній процес. Кодери та декодери можуть бути реалізовані програмно або апаратно на ПЛІС(програмована логічна інтегральна схема). Сучасні ПЛІС, наприклад FPGA(Field-programmable gate array), можуть

реалізовувати до мільйона LUT(Lookup Table), на яких може бути реалізована будь-яка булева функція від 6-ти змінних.

1.3 Алгоритм Брона-Кербоша

На даний момент, одним із ефективних способів пошуку множини максимальних клік на графі є алгоритм Брона-Кербоша[4]. Який, однак, не може бути розпаралелений, що є його недоліком зважаючи на сучасну тенденцію збільшення швидкодії обчислювальної техніки за рахунок багатоядерності. Алгоритм був розроблений голландськими математиками Броном та Кербошем у 1973 році. Під час пошуку максимальної кліки використовується той факт, що будь-яка кліка є максимальним повним підграфом. Починаючи з найпершої вершини, яка являє собою повний підграф, алгоритм на кожному кроці намагається вже побудований підграф збільшити додаючи в нього нові вершини. Цей алгоритм є рекурсивною процедурою(рис 1.4, рис 1.5) яка оперує трьома множинами вершин:

- compsub – множина, що містить повний підграф на певному кроці алгоритму.
- candidates – множина вершин, які потенційно можуть бути частиною підграфу compsub.
- not – множина вершин, які вже були використані для збільшення compsub.

```
ПРОЦЕДУРА extend(candidates, not):
   ПОКИ candidates НЕ порожньо І not НЕ містить вершини, з'єднаної з усіма вершинами з candidates,
   ВИКОНУВАТИ:
   1 Вибираємо вершину v з candidates і додаємо її в compsub
   2 Формуємо new_candidates і new_not, видаляючи з candidates і not вершини, не З'ЄДНАНІ  з v
   3 ЯКЩО new_candidates і new_not порожні
   4 ТО compsub - кліка
   5 ІНАКШЕ рекурсивно викликаємо extend(new_candidates,new_not)
   6 Видаляємо v з compsub і candidates і поміщаємо в not
```

Рис. 1.4. Псевдокод алгоритму Брона-Кербоша(Зробити кресленняблок схему).

Складність алгоритму лінійна відносно кількості клік у графі та було показано, що в найгіршому випадку він працює за:

$$O(3^{n/3}) , \qquad (10)$$

де n – кількість вершин у графі[5]. Алгоритм знаходить всі повні кліки графа, серед яких уже вибирається максимальна.

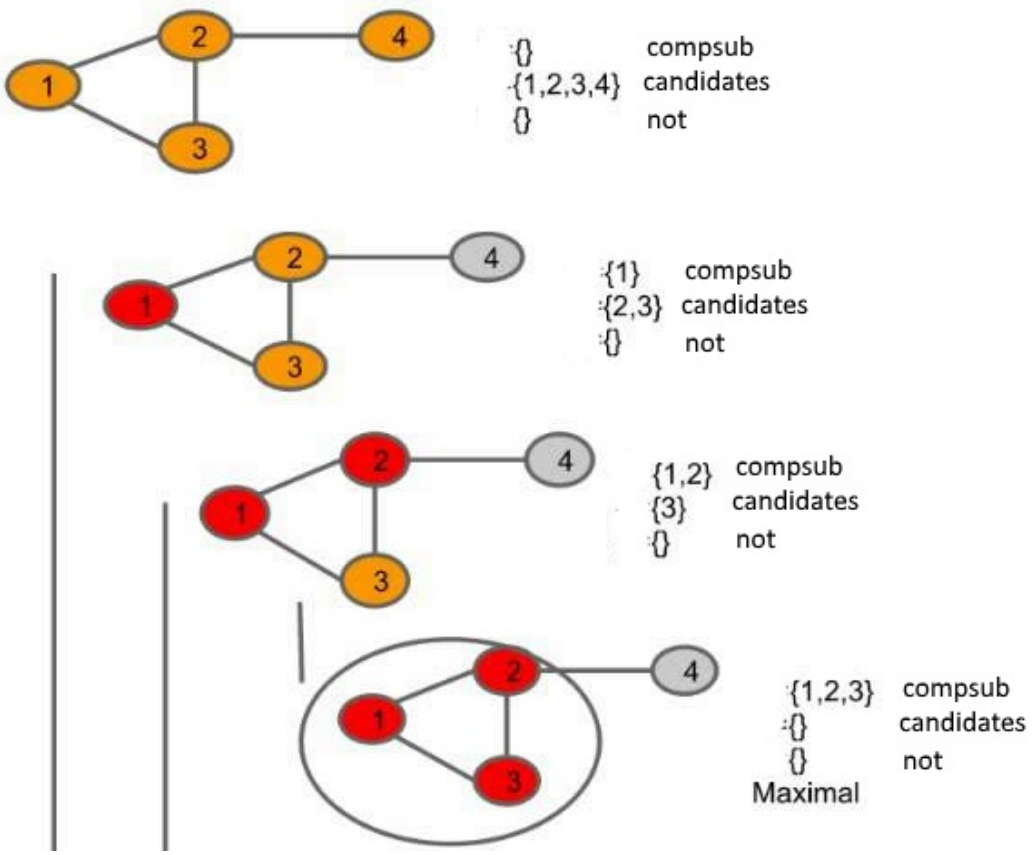


Рис. 1.5. Візуалізація роботи алгоритму Брона-Кербоша.

Реалізація процедури на мові програмування Java зображена на рисунку 1.6. Де r – множина вершин кліки на певному кроці, p - множина вершин, які можуть потрапити у кліку, x - множина вже перевірених вершин.

```java
private void findAllCliques(HashSet<Node> r, HashSet<Node> p, HashSet<Node> x) {
    if (p.isEmpty() && x.isEmpty()) {
        cliques.add(r);
    }
    Iterator<Node> nodeIterator = p.iterator();
    while (!p.isEmpty() && nodeIterator.hasNext()) {
        Node node = nodeIterator.next();
        HashSet<Node> singleton = new HashSet<>();
        singleton.add(node);
        findAllCliques(union(r, singleton),
                       intersection(p, graph.getChildren(node)),
                       intersection(x, graph.getChildren(node)));
        p = difference(p, singleton);
        x = union(x, singleton);
        if (p.isEmpty()) {
            nodeIterator = p.iterator();
        }
    }
}
```

Рис. 1.6. Рекурсивна реалізація алгоритму Брона-Кербоша на мові Java.

1.4 Визначення нероздільних кодів.

Для пошуку нероздільних кодів у даній роботі було використано особливості графа Хемінга(рис 1.6) та алгоритму Брона-Кербоша. По суті, кліка графа Хемінга і є кодом. Якщо задати мінімальну відстань Хемінга для графа, наприклад d=3, то всі кліки у такому графі будуть нероздільними кодами. Кожна кліка буде містити t елементів, а розрядність коду можна задати довільну. В даній роботі вирішується задача підвищення аналітичної швидкості завадостійких кодів. Аналітична швидкість коду C дорівнює:

$$(\log t)/n, \qquad (11)$$

де t - кількість кодослів у коді, n - кількість розрядів кодослова. Величина (11) по суті показує відношення кількості реально переданої інформації до фактично переданої. Отже в даному випадку нас цікавить максимальна кліка(з найбільшою кількістю кодослів) для максимальної аналітичної швидкості коду. Наприклад, для графа зображеного на рисунку 1.7 є декілька максимальних клік. Наприклад одна з них складається кодом *C(4,3,2)* і складається з кодослів:

001, 010, 100, 111. (12)

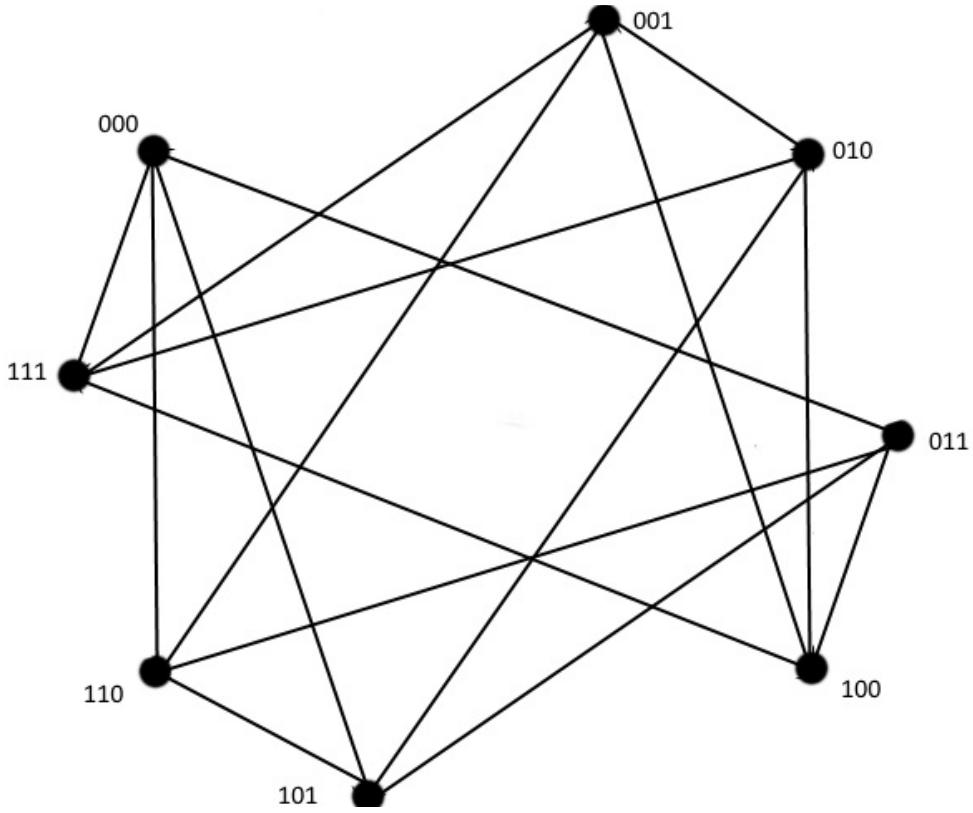


Рис. 1.7 Граф Хемінга із кодослів розрядності 3 та d≥2.

В даній роботі буде розглянуто 8-ми та 9-ти розрядні коди, для виправлення одиночних та подвійних помилок.

2. УДОСКОНАЛЕННЯ АЛГОРИТМУ БРОНА-КЕРБОША

2.3 Еквівалентність кодів.

Для оптимізації алгоритму, при виконанні задачі пошуку кодів, скористаємося поняттям еквівалентності кодів. А саме еквівалентності по xor та по перестановці.

Коди C та C^* називають еквівалентими по xor, якщо виконується умова:

$$\begin{aligned} &c_1 xor\ c_x = c_1^* \\ &c_2 xor\ c_x = c_2^* \\ &\ldots\ldots\ldots\ldots \\ &c_n xor\ c_x = c_n^*, \end{aligned} \qquad (13)$$

де c_1-c_n кодослова які належить коду C, c_1^*-c_n^* - кодослова які належить коду C^*, c_x – будь-яке кодослово яке належить коду C.

Далі на прикладі коду (12) показано побудову для нього еквівалентного коду, де за c_x вибрано перше кодослово: 001.

$$\begin{aligned} &001\ xor\ 001 = 000 \\ &010\ xor\ 001 = 011 \\ &100\ xor\ 001 = 101 \\ &111\ xor\ 001 = 110 \end{aligned}$$

Отже отриманий код *000, 011, 101, 110* є еквівалентним по *xor* коду (12), та є клікою на графі (рис 1.6). До того ж, для нього зберігається така ж кодова відстань як і для початкового коду:

$$c_1' xor\ c_2' = (c_1 xor\ c_x)\ xor\ (c_2 xor\ c_x) = c_1 xor\ c_2 \qquad (14)$$

Для даного коду можна побудувати ще 3 еквівалентні коди, які будуть включати кодослово *000*. Суть цієї еквівалентності полягає в тому, що якщо для певного коду не можливо додати нового кодослова, то його не можна додати і до всіх еквівалентних кодів.

В загальному випадку алгоритм Брона-Кербоша перебирає K(кількість вершин) варіантів початку аналізу, в яких кожного разу аналіз починається з

нової вершини. У випадку графа Хемінга, позначення вершин двійковими числами природне, так як об'єктами вершин графа і є ці ж числа. Враховуючи те, що всі максимальні коди еквівалентні по XOR, кодам які починаються з нульової вершини, можна обмежити алгоритм так, щоб він починався лише з нульової вершини. Таким чином ми позбудемося *K-1* початків аналізу, фактично пришвидшивши пошук кліки у *K* разів. Для обґрунтування цього підходу, потрібно припустити що існує максимальна кліка, яка не містить нульовго кодослова. Тоді будь-яке кодослово, що належить до цієї кліки, може перетворити її по *xor* на таку, яка міститиме нульове кодослово. І вона буде знайдена алгоритмом, тоді коли починати пошук тільки з нульового кодослова.

Перетворення коду по перестановці – створення новго коду шляхом перестановки однакових розрядів кожного кодослова у коді. Перестановка задається матрицею *2xN*, або просто одним рядком довжиною *N,* де *N* – кількість розрядів коду. Нехай існує деяка матриця перестановок(рис 1.7) відповідно до якої, нульовий розряд коду буде переміщено на позицію 1, перший розряд – позиція 2 та другий розряд – позиція 0. Варто зауважити, що перестановка не змінює кодослова яке складається лише з одиниць або нулів, та кодова відстань після перестановки зберігається.

0	1	2
1	2	0

Рис. 2.1 Приклад задання перестановки.

Якщо застосувати вищерозглянуту перестановку до коду (12) то буде отримано новий код:

001 = 100
010 = 001
100 = 010
111 = 111

Два коди називаються еквівалентними по перестановці, якщо існує така перестановка, яка перетворює один код на другий. Технічно таку операцію можна реалізувати в три етапи:

- Перетворити код у двохвимірний масив.
- Переміщення стовпців масиву відповідно до перестановки.
- Перетворення масиву назад у код.

Таким чином можна вибрати друге кодослово, яке повинне бути присутнє в коді, це має бути будь-яке кодослово кількість одиниць в якому, дорівнює або більша за *d*. Тобто починати пошук кліки із двох уже відомих вершин: нульової та такої в якій кількість одиниць дорівнює *d,* для першого варіанту початку пошуку. Пошук кліки на вищезгаданому графі(рис 1.6). буде починатись із кодів *000* та *011*. Для наступних варіантів початку можна брати друге кодослово з кількістю одиниць *d+1, d+2* і т.д. Варіанти з однаковою кількістю одиниць, які розміщені на різних позиціях у кодослові, розглядати не потрібно, бо одержаний код однак буде максимальним по перестановці. Тому що перше кодослово з усіма нулями перестановка не змінює.

2.4 Алгоритм визначення кодів більшої розрядності на базі кодів меншої розрядності

Цей алгоритм дозволяє, у деяких випадках, з уже визначеного максимального коду *n* розрядності, створити максимальний код розрядності *n+1*. Нехай є максимальний *n*-розрядний код *C(t,n,d)*. На першому кроці алгоритму потрібно створити множину *m1,* яка буде містити всі кодослова

коду *C* та відповідні двійкові числа з відстанню Хемінга *d-2* до кодослів коду. Потім створюється множина *m2,* яка є перетином множин *m1* та V^n. Так як при формуванні множини *m1* до неї були включені всі можливі числа з відстанню Хемінга *d-2* від кодослів коду *C(t,n,d)*, то відстань будь-якого числа з *m2* до кодослова з *C(t,n,d)* не менша *d-1*. Після цього, потрібно запустити алгоритм Брона-Кербоша на множині *m2*, побудувавши граф Хемінга перед цим. Буде одержано максимальний код $C(t_1,n,d)$. Очевидно, що кодова відстань між цими кодами $C(t_1,n,d)$ та *C(t,n,d)* дорівнює *d-1*. Щоб побудувати *n+1* розрядний код, потрібно додати до всіх кодослів коду *C(t,n,d)* одиницю після *n*-го розряду, та аналогічно нуль до кодослів коду $C(t_1,n,d)$. Таким чином збільшуючи кодову відстань ще на 1, і в результаті вона дорівнює *d*. Якщо отриманий код, матиме таку ж кількість кодослів як і початковий $(t_1=t)$, то об'єднавши ці коди буде отримано новий код *C(2t,n+1,d)*.

3. ПРОГРАМНИЙ КОМПЛЕКС ДЛЯ ВИЗНАЧЕННЯ ЗАВАДОСТІЙКИХ КОДІВ

3.3 Загальна структура комплексу

Програмний комплекс для визначення максимальних нероздільних завадостійких кодів виконаний мовою програмування Java. Для збірки проекту використовується *gradle* – система автоматичного збирання проектів, яка використовує предметно-орієнтовну мову, замість традиційної XML-подібної. Таким чином, щоб зібрати та запустити проект в практично будь-якому середовищі, достатньо виконати команду *gradle run*, всі залежності будуть автоматично завантажені у проєкт.

Проєкт на мові Java включає дві основні директорії *src/main* та *src/test* в яких містяться вихідні коди власне програмної системи та тестів відповідно. Також, вибрана мова програмування надає систему пакетів та модулів, для відокремлення різних структурних частин проекту. Модулі фактично є самостійними проектами, які можуть бути використані при реалізації інших проектів.

Розроблений в даній роботі комплекс є одним самостійним модулем, який складається з двох основних частин: *main* та *test.* Загальна структура частин та їх складових(пакетів та додаткових ресурсів) зображена на рисунку 3.1. Найменшою структурною одиницею є клас, який зазвичай розміщено в окремому файлі. На загальних ілюстраціях класи позначені не будуть.

Для реалізації графічного інтерфейсу користувача було обрано вбудовані в Java(починаючи з версії 8) засоби, а саме набір інструментів JavaFX, який дозволяє створювати насичені графічні інтерфейси для десктопу, мобільних пристроїв та веб-додатків.

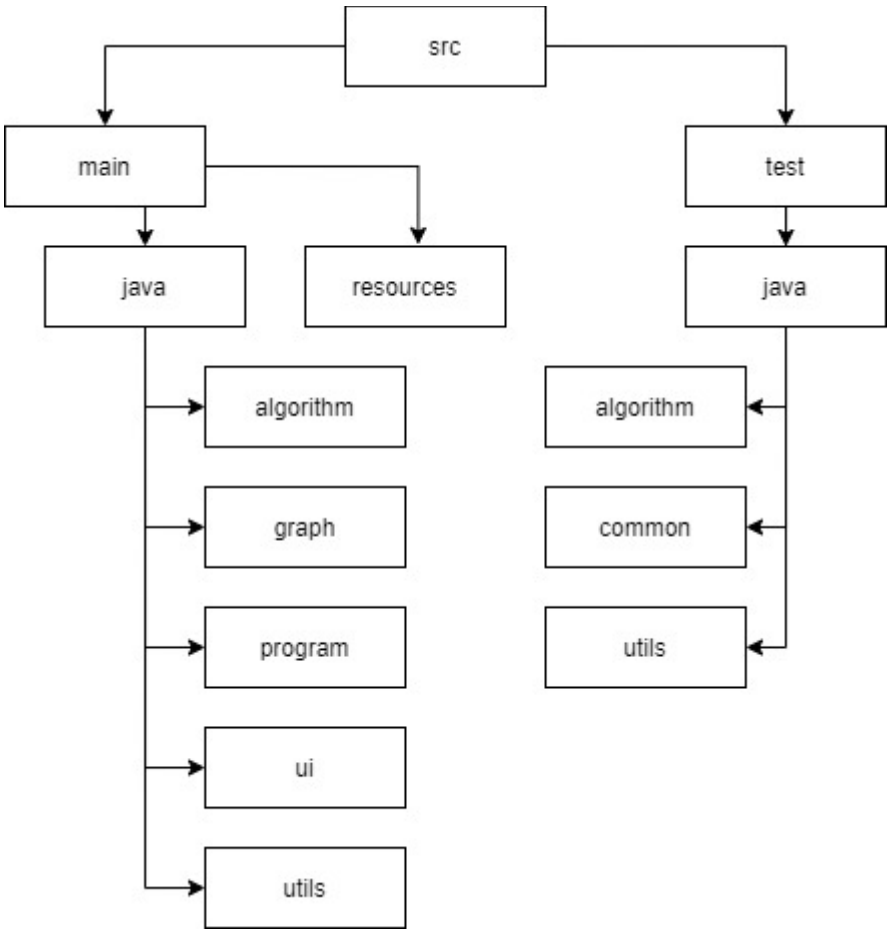


Рис. 3.1. Загальна структура комплексу.

3.4 Основні пакети.

Далі приведено опис основної(*main*) частину проекту(рис. 3.2), його класи та основні методи.

Пакет *program.*

Основний пакет з якого починається запуск програми.

Клас *Main* – наслідує JavaFX клас *Application*. Містить такі методи:

- *main* – головний метод з якого починається запуск програми.
- *start* – унаслідуваний від метод, який *Application* заванатажує та показує вікно для користувача.

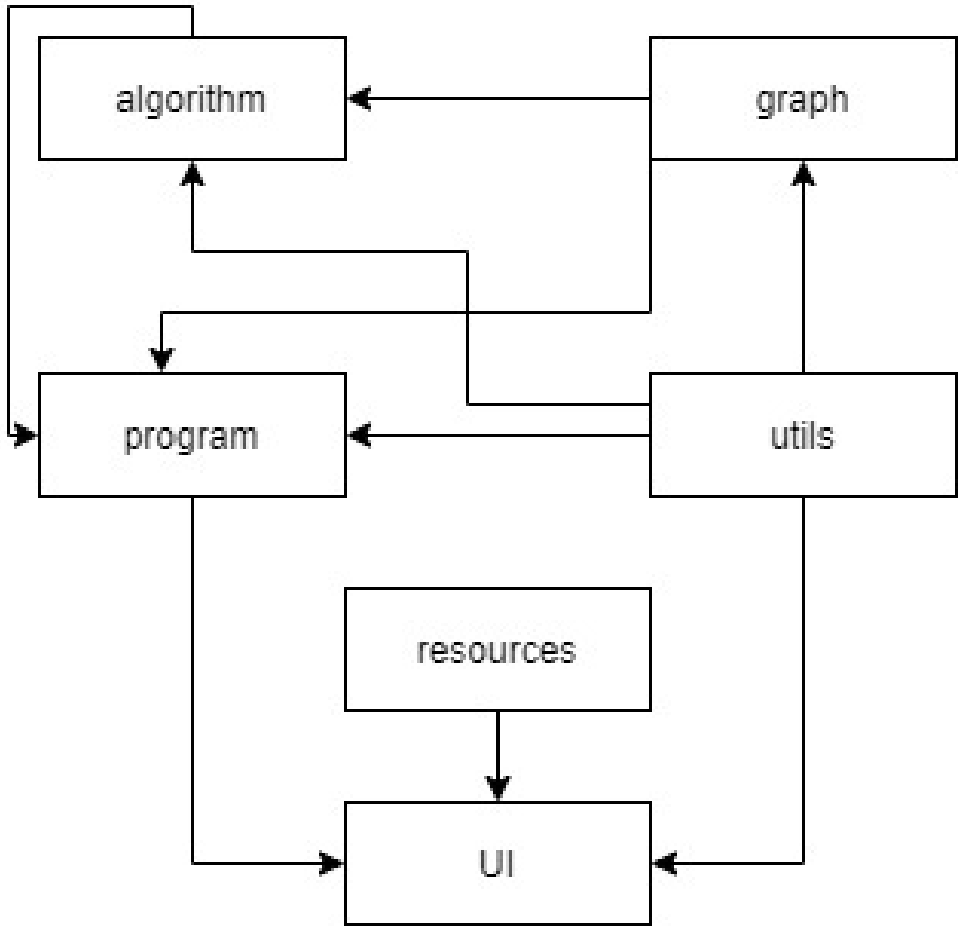


Рис. 3.2. Загальна схема залежностей пакетів та ресурсів.

Клас *CodeResolver* – реалізує стандартний інтерфейс *Callable*, для можливості запуску алгоритму в окремому потоці. Мова Java надає досить широкий набір інструментів для реалізації багатопічних за стосунків, одни із яких і є *Callable*. В даній роботі це необхідно для надання користувачу можливості збереження проміжних даних з якими працює алгоритм, для подальшого продовження виконання, оскільки пошук може виконуватись дуже довго. Основні задачі які виконує *CodeResolver* – це підготовка алгоритму до запуску за відповідних умов, зупинка алгоритму у разі потреби зі збереженням даних та обробка отриманих результатів.

Містить такі основні методи:

- *findNewCodes(int,int)* – підготовка до запуску пошуку нового коду на новому графі, де розрядність коду та відстань Хемінга задається параметрами.

- *loadAndContinueSearch()* – підготовка до запуску пошуку коду, коли алгоритм було зупинено, завантажує збережені проміжні дані для продовження пошуку.

- *stopExecuting()* – метод який посилає сигнал зупинки алгоритму.

- *wasInterrupted()* – повертає значення істини тоді, коли алгоритм було зупинено користувачем.

- *call()* – метод інтерфейсу *Callable*, який власне і запускає алгоритм та повертає результати обчислень, надає можливість виконати це в окремому потоці.

Пакет *graph.*

Цей пакет містить реалізацію неорієнтовного графа та в особливості графа Хемінга і методи для його побудови. Та надає певну функціональність для роботи з графом – додавання вершин тощо.

Клас *Node* – являє собою вершину графа в якій зберігається цілочисельне значення, тобто в нашому випадку кодослово.

Клас *Edge* – представляє ребро графа, та містить посилання на дві вершини між якими існує це ребро.

Клас *Graph* – представляє власне граф, який містить список ребер та вершини. Методи для додавання вершин, та ребер, та отримання тих самих вершин та ребер графу. Метод для пошуку всіх вершин графу, які суміжні до певної вершини(рис 3.3).

```
public HashSet<Node> getChildren(Node node) {
    HashSet<Node> children = new HashSet<>();
    edges.forEach(edge -> { if (edge.getX().getCode() == node.getCode()) { children.add(edge.getY()); }
                            else if (edge.getY().getCode() == node.getCode()) children.add(edge.getX()); });
    return children;
}
```

Рис. 3.3.Метод пошуку суміжних вершин.

Клас *HammingGraph* – представляє граф Хемінга, для якого задається розрядність кодів та кодова відстань.

Клас *GraphBuilder* – містить функціональність для побудови графа Хемінга, а саме статичний метод *makeHammingGraph*(рис 3.4). Спочатку визначається максимальне значення коду певної розрядності(*maxValue*). Створюється об'єкт графу Хемінга, та виконується перебір всіх чисел від 0 до *maxValue*, для кожного з яких створюється вершина(*Node*). Для кожної вершини перевіряється відстань Хемінга із уже присутніми вершинами у графі, якщо вона не менша за задану, то ці вершини з'єднуються ребром.

```
public static HemmingGraph makeHammingGraph(int digits, int minDigits) {
    int maxValue = (int) Math.pow(2, digits);
    HemmingGraph g = new HemmingGraph(digits, minDigits);
    for (int i = 0; i < maxValue; i++) {
        Node node = new Node(i);
        g.getNodes().forEach(nd -> {
            if (hemmingDistance(node.getCode(), nd.getCode(), digits) >= minDigits)
                g.addEdge(new Edge(node, nd));
        });
        g.addNode(node);
    }
    return g;
}
```

Рис. 3.4. Метод побудови графу Хемінга.

Пакет *utils.*

Додатковий пакет, який містить допоміжні класи з додатковою функціональністю, яка досить часто використовується в різних частинах проекту.

Клас *Sets* – реалізує функції роботи з множинами, на основі стандартних об’єктів *HashSet*, які передбачають унікальність елементів. Операції над множинами використовуються в самому алгоритмі Брона-Кербоша. Методи класу *Sets*:

- *union(a,b)* – об’єднання двох множин.
- *intersection(a,b)* – перетин множин.
- *difference(a,b)* – різниця множин.
- *differenceLinkedSet(a,b)* – різниця множин, але результуючим є об’єкт *LinkedHasSet* в якому зберігається порядок елементів, у якому вони були додані.
- *toSet*– перетворення списку(*List*) на *HashSet*.

Клас *Serializer* – містить методи серіалізації та десеріалізації об’єктів. Серіалізація – це процес перетворення структури даних у потік байтів. Десеріалізація – зворотний процес, тобто відновлення стану об’єкту із потоку байт. Мова Java надає стандартні інструменти для виконання вищезгаданих операцій, а саме методи *readObject* та *writeObject*. Для того, щоб певний об’єкт міг бути серіалізований або десеріалізований, клас цього об’єкту повинен реалізувати інтерфейс *Serializable.*

Serializer виконує ці операції та наступний запис у файл або читання з файлу та містить методи для збереження та завантаження об’єктів графу та проміжних результатів алгоритму. Використовується для збереження результатів після зупинки алгоритму та їх відновлення для продовження роботи алгоритму.

Клас *CodeUtils* – містить функції для роботи з кодами(кодословами), які представлені чілочисельними значеннями. Методи класу *CodeUtils*:

- *numberWeight* – повертає вагу цілочисельного значення, тобто кількість одиничних розрядів, шляхом побітового зсуву та перевірки кожного біту(рис 3.5).

```
public static int numberWeight(int num, int size) {
    int weight = 0;
    for (int i = 0; i < size; i++) {
        if ((num >> i & 1) > 0) {
            weight++;
        }
    }
    return weight;
}
```

Рис. 3.5. Метод визначення кількості одиниць у числі.

- *hemmingDistance* – повертає відстань Хемінга між двома числами (рис 3.6). Фактично виконує операцію між цими числами XOR, а потім підраховує одиничні розряди.

```
public static int hemmingDistance(int a, int b, int size) {
    return numberWeight(a ^ b, size);
}
```

Рис. 3.6. Метод визначення відстані Хемінга між числами.

- *toBinary* – перетворення числа у масив із 0 та 1.
- *makeIntWithWeight* – створення цілого числа з певною кількістю одиниць у молодших двійкових розрядах.
- *convertToCode* – перетворює множину клік у множину об’єктів типу Code.

Клас *Code* – представляє собою завадостійкий код, який складається із деякої кількості кодослів, зберігає розрядність коду та всі кодослова, як список цілих чисел та містить реалізацію операцій які можна виконати над кодом:

- *toMatrix* – виконує перетворення коду у матрицю цілочисельних значень 1 або 0 (рис 3.7). Рядками матриці є кодослова.

```
public int[][] toMatrix() {
    int[][] matrix = new int[getSize()][];
    int i = 0;
    for (int code : codes) {
        int[] array = CodeUtils.toBinary(code, bitRate);
        matrix[i++] = array;
    }
    return matrix;
}
```

Рис. 3.7. Метод перетворення коду у матрицю.

- *sort* – сортування код ослів за зростанням.

- *getDistance()* – знаходить мінімальну кодову відстань для даного коду(рис 3.8). Потрібно виконати перебір всіх кодослів і порівняти кодову відстань з усіма іншими кодословами.

```
public int getDistance() {
    int minimal = bitRate;
    for (int code1 : codes) {
        for (int code2 : codes) {
            if (code1 != code2) {
                int distance = CodeUtils.hemmingDistance(code1, code2, bitRate);
                if (distance < minimal) {
                    minimal = distance;
                }
            }
        }
    }
    return minimal;
}
```

Рис. 3.8. Метод визначення мінімальної кодової відстані.

- *getDistance(int)* – метод визначення кодової відстані конкретного кодослова до всього коду.

Пакет *algorithm.*

В даному пакеті знаходиться реалізація удосконаленого алгоритму Брона-Кербоша.

Клас *IntermediateResult* – призначений для збереження всієї інформації виконання алгоритму на певному кроці. Такою інформацією є сам граф Хемінга, множина уже знайдених клік та стеки всіх множин якими оперує алгоритм. Під час зупинки алгоритму створюється об'єкт *IntermediateResult* який серіалізуєтьсь та записується у файл. Під час продовження роботи алгоритму, цей об'єкт читається з файлу та десеріалізується.

Клас *BronKerboshAlgorithm* – містить власне реалізацію алгоритму та допоміжні дані. Основний інтерфейсний метод який надає цей клас для пошуку клік називається *findMaxCliques* (рис 3.9). Все відбувається у два етапи: спочатку запуск алгоритму пошуку, а потім пошук максимальних клік, та видалення менших, які не є максимальними.

```
public HashSet<HashSet<Node>> findMaxCliques() {
    findAllCliques();
    filterCliques();
    return cliques;
}
```

Рис. 3.9. Метод визначення множини максимальних клік.

Алгоритм Брона-Кербоша реалізовано ітеративно, для цього було використано стеки для всіх множин якими оперує алгоритм, та які, в традиційній(рекурсивній) реалізації передаються як параметри. Тобто замість рекурсивного спуску, виконується запис параметрів у стеки, а у випадку рекурсивного повернення читання цих параметрів зі стеків.

Таке рішення має деякі переваги. По-перше, було забезпечено неможливість переповнення стеку. Так, як у мові Java під стек виділяється зазвичай 1 мегабайт пам'яті, то на великих графах цього стеку може не

вистачити. По-друге, це дозволяє виконувати збереження проміжних результатів, та продовження виконання алгоритму з певної ітерації, чого при рекурсивній реалізації зробити не можливо. Тому що, стек викликів та їх параметри під час виконання, програмісту не доступні, крім того, не існує можливості почати виконання функції із наперед заданим стеком викликів. Проте існують і недоліки. Реалізація стала складнішою(рис 3.10) у порівнянні з вищезгадам рекурсивним варіантом та не такою читабельою.

Алгоритм виконується доки множина потенційних вершин кліки або множина кліки не порожні. На кожній ітерації перевіряється прапорець *stop* який вказує на зупинку алгоритму користувачем. У випадку зупинки виконується збереження всіх проміжних даних у змінну в класі.

Прапорець *stop* має тип *volatile boolean*. Ключове слово *volatile* означає, що змінна може бути змінена ззовні і програма буде звертатись до тієї змінної кожного разу при доступі. Тому, що компілятор мови *Java* зазвичай виконує оптимізації, та підставити замість змінної, її значення, якщо не буде виявлено що вона змінюється. В даному випадку початкове значення змінної дорівнює *false* і в самому алгоритмі воно не змінюється. Змінюється значення із зовнішнього потоку, тому потрібно вказати компілятору, що для цієї змінної не потрібно виконувати оптимізації.

Також додана змінна *depth*, яка по суті є лічильником глибини імітованої рекурсії. На кожному кроці перевіряється її значення, та значення відповідної вершини яка розглядається. Суть оптимізації полягає в тому, що перші два кодослова для коду вже вибрані та збережені на початку множини *candidates*, розглядати наступні варіанти початку алгоритму не потрібно.

```
private void findAllCliques() {
    HashSet<Node> compsub = compsubStack.pop();
    HashSet<Node> candidates = candidateStack.pop();
    HashSet<Node> not = notStack.pop();
    int depth = compsubStack.size();
    while (!candidates.isEmpty() || !compsub.isEmpty()) {
        if (stop) {
            saveIntermediateResult();
            return;
        }
        HashSet<Node> singleton = new HashSet<>();
        if (!candidates.isEmpty()) {
            depth++;
            Node node = candidates.iterator().next();
            if (depth == 2) {
                if (CodeUtils.numberWeight(node.getCode(),
                                           graph.getBitRate()) != graph.getDistance()) {
                    break;
                }
            }
            singleton.add(node);

            compsubStack.push((HashSet<Node>) compsub.clone());
            candidateStack.push(candidates);
            singletonStack.push(singleton);
            notStack.push(not);

            compsub.addAll(singleton);

            HashSet<Node> notConnected = difference(toSet(graph.getNodes()),
                                                    graph.getChildren(node));
            candidates = differenceLinkedSet(candidates, notConnected);
            candidates = differenceLinkedSet(candidates, singleton);
            not = difference(not, notConnected);
        } else {
            depth--;
            if (not.isEmpty()) {
                cliques.add(compsub);
            }
            compsub = compsubStack.pop();
            candidates = candidateStack.pop();
            singleton = singletonStack.pop();
            not = notStack.pop();
            candidates = difference(candidates, singleton);
            not.addAll(singleton);
        }
    }
    stop = true;
}
```

Рис. 3.10. Ітеративна реалізація удосконаленого алгоритму Брона-Кербоша.

Що і забезпечує перевірка глибини рекурсії та значення поточної вершини. Якщо глибина дорівнює 2, а значення поточної вершини не

дорівнює заздалегідь вибраному другому кодослову, у якому кількість одиниць дорівнює *d* то продовжувати пошук не потрібно. В інакшому випадку значення поточних множин записуються у відповідні стеки: *compub* у *compsubStack, not* у *notStack, candidate* у *candidateStack* та поточна вершина у *singletonStack.* Виконуються відповідні операції з цими множинами і результати записуються у локальні змінні.

Так відбувається доки множина *candidates* не стане порожньою, це означає, що більше нема вершин для перегляду. Перевіряється множина *not,* якщо вона порожня то алгоритм знайшов кліку. Зі стеку беруться значення множин на попередньому кроці та присвоюються відповідним локальним змінним, таки чином імітуючи рекурсивний підйом і алгоритм продовжує своє виконання.

Так виглядає реалізація пошуку усіх клік. Потім виконується фільтрація клік, тобто видалення не максимальних, у випадку якщо вони були знайдені(рис 3.10).

```
private void filterCliques() {
    Iterator<HashSet<Node>> cliqueIterator = cliques.iterator();
    if (!cliqueIterator.hasNext()) {
        return;
    }
    HashSet<Node> max = cliqueIterator.next();
    while (cliqueIterator.hasNext()) {
        HashSet<Node> next = cliqueIterator.next();
        if (next.size() > max.size()) {
            max = next;
        }
    }
    final int maxSize = max.size();
    cliques.removeIf(c -> c.size() != maxSize);
}
```

Рис. 3.10.Метод видалення не максимальних клік.

Для старту алгоритму необхідно створити об’єкт *BronKerboshAlgorith* предавши йому граф. Потім виконується ініціалізація стеків. Є два варіанти, початкова *initStacks*(рис 3.11), яка задає початкові значення алгоритму відповідно до заданного граф та *withIntermediateResult* , яка встановлює задані проміжні результати для продовження пошуку.

```
public BronKerboshAlgorithm initStacks() {
    compsubStack.add(new HashSet<>());
    HashSet<Node> nodes = new LinkedHashSet<>();
    Node zero = makeNode(0);
    Node second = makeNode(graph.getDistance());

    nodes.add(zero);
    nodes.add(second);
    for (Node nd : graph.getNodes()) {
        if (nd.getCode() != zero.getCode() && nd.getCode() != second.getCode()) {
            nodes.add(nd);
        }
    }
    candidateStack.add(nodes);
    notStack.add(new HashSet<>());
    return this;
}
```

Рис. 3.11.Ініціалізація стеків.

При старті нового пошуку, під час ініціалізації стеків, у початкову множину *candidates* задається спочатку нульова вершина, а потім вершина у якої кількість одиниць дорівнює *d*. Таким чином алгоритм починає пошук із цих двох вершин, та зупиняється при виборі у якості початкової іншої.

3.5 Графічний інтерфейс користувача.

Як уже зазначалося, графічний інтерфейс був виконаний із використанням JavaFX. Цей інструмент досить гнучкий та надає багато можливостей. Існують додатки для сучасних IDE, які дозволяють конструювати візуальні вікна та додавати в них елементи та відразу бачити результат. Також є можливість створювати вікна та елементи в самому коді або задавати конфігурацію цих вікон та їх елементів в окремих *fxml* файлах, які мають структуру XML.

Структура графічного застосунку зображена на рисунку 3.12.

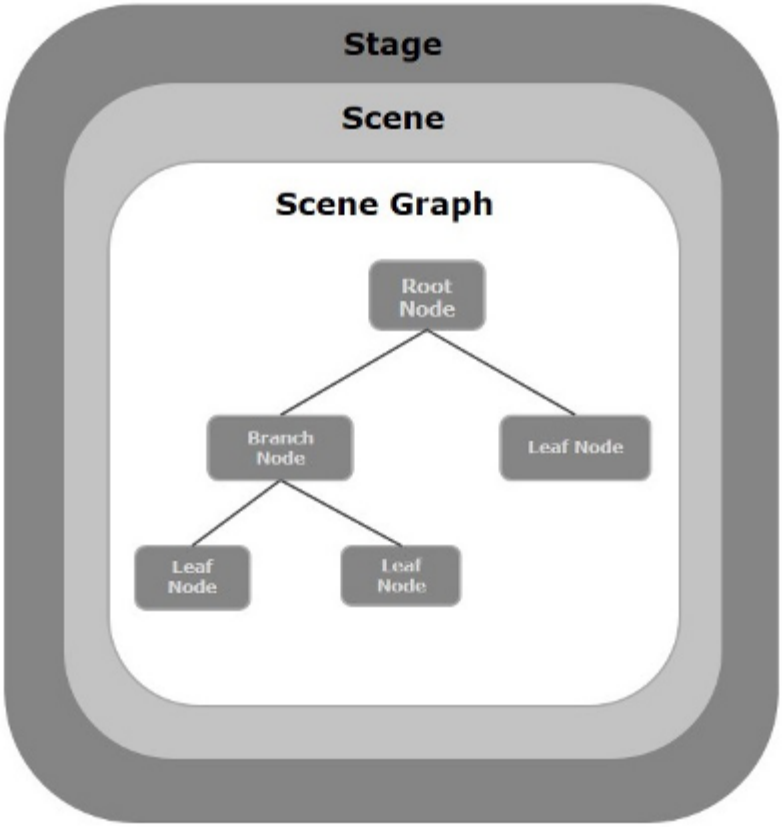


Рис. 3.12. Загальна структура графічного застосунку.

Елемент *Stage* – представляє собою власне вікно програми, яке створюється при запуску. Розмір вікна можна задати, як розмір *Stage*.

Елемент *Scene* – об’єкт, що містить у собі все наповнення вікна.

Root Node - початковий елемент *Scene.* Представляє собою дерево, листками та гілками якого є певні графічні елементи, які будуть відображені на екрані, наприклад кнопки, текстові вікна тощо.

Така програма працює у безкінечному циклі, доки не буде завершена користувачем.

Пакет *ui*.

Пакет який містить реалізацію графічного інтерфейсу та контролерів які допомагають запускати логічні частини програми розміщені в інших пакетах, обробляти результати їх виконання та реагувати на дії користувача. По суті, контролер пов'язує графічну частину програми з алгоритмічною. Обробляє введені користувачем дані, та передає їх у алгоритми і так само отримує результати виконання певних алгоритмів і відображає результати користувачу. Такий підхід називається MVC(*Model-View-Controller)*. MVC(рис 3.13) – це шаблон розробки програмного забезпечення, який передбачає розділення бізнес-логіки та графічного інтерфейсу. Таким чином можна використовувати інструменти JavaFX, як *View* задаючи контролери не залежно він бізнес-логіки, в даній роботі це алгоритм Брона-Кербоша та інші дпоміжні функції.

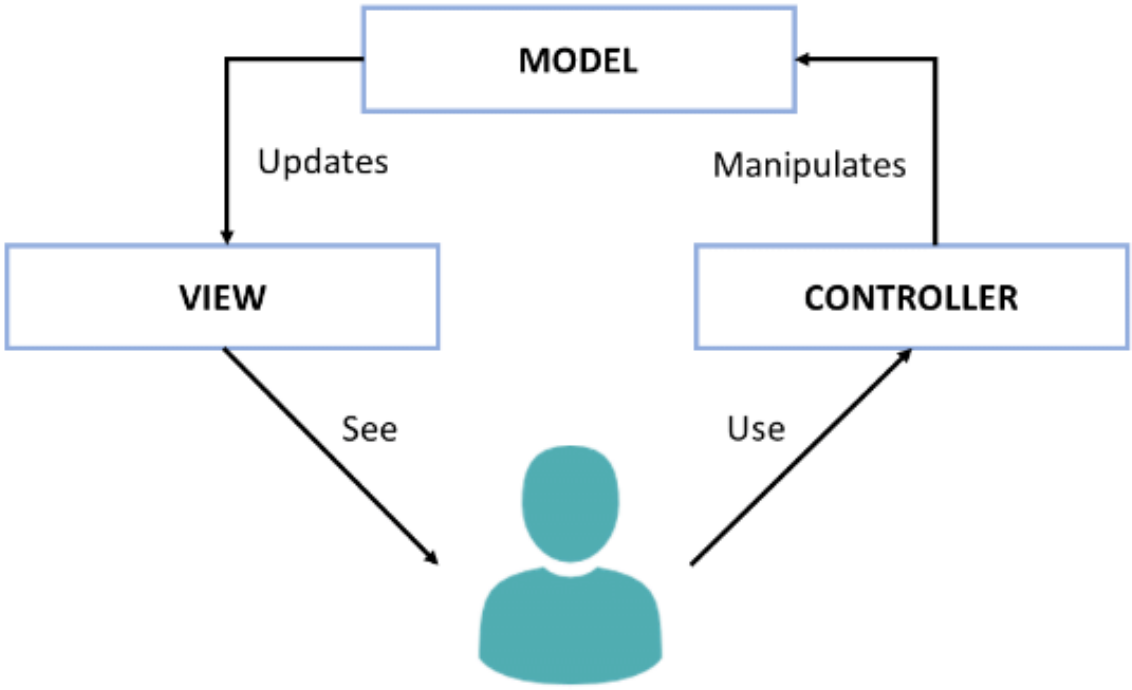


Рис. 3.13. Взаємодія компонентів MVC.

Основні вікна програми розміщені у файлах *fxm*(рис 3.14) в дерикторії resources. Як видно з рисунка у ткаих файлах можна дуже просто задавати різні параметри вікна, додавати елементи. Сама структура XML відповідає структурі застосунку, яке має вигляд дерева В даній роботі таких вікна є три:

- find-new-code.fxml – вікно для пошуку нового коду із заданими параметрами
- load-code.fxml – вікно для завантаження збережених результатів та продовження пошуку.
- main-window.fxml – головне вікно програми, з якого починається запуск.

Також є вікна які конфігуруються та створюються динамічно:

- вікно для виведення помилки та повідомлення.
- вікно для виведення коду та подальшої його обробки у разі потреби.
- вікно для відображення результатів певних операцій над кодом.

Основні класи.

Клас *WindowManager* – відповідає за створення нових вікон – завантаження їх з файлів або генерацію під час виконання програми.

- Метод *showCodeWindow* – генерує спеціальне вікно для відображення коду у двійковому вигляді. Розмір вікна залежить від розрядності коду та кількості кодослів.

main > resources > main-window.fxml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <?import javafx.scene.control.*?>
4   <?import javafx.scene.layout.*?>
5
6   <GridPane hgap="10" prefHeight="231.0" prefWidth="223.0" vgap="10"
7             xmlns="http://javafx.com/javafx/10.0.2-internal"
8             xmlns:fx="http://javafx.com/fxml/1"
9             fx:controller="ui.MainWindowController">
10     <Label alignment="CENTER_RIGHT" contentDisplay="CENTER"
11            prefHeight="16.0" prefWidth="127.0" GridPane.columnIndex="2"
12            GridPane.rowIndex="4">Code Name</Label>
13     <Button prefHeight="26.0" prefWidth="172.0" GridPane.columnIndex="2"
14             GridPane.rowIndex="7" onAction="#startNewButtonClick">Start new</Button>
15     <Button prefHeight="26.0" prefWidth="173.0" GridPane.columnIndex="2"
16             GridPane.rowIndex="8" onAction="#loadCodeButtonClick">Load</Button>
17     <TextField fx:id="codeName" GridPane.columnIndex="2" GridPane.rowIndex="5" />
18
19  </GridPane>
20
```

Рис. 3.14. Вміст файлу main-window.fxml.

- Метод *showSimpleInfoWindow*(рис 3.15) – створює вікно для виведення невеликої за обсягом інформації.

```
public static void showSimpleInfoWindow(String title, String msg) {
    Label secondLabel = new Label(msg);
    StackPane secondaryLayout = new StackPane();
    secondaryLayout.getChildren().add(secondLabel);
    Scene secondScene = new Scene(secondaryLayout, 200, 100);
    Stage window = new Stage();
    window.setTitle(title);
    window.setScene(secondScene);
    window.show();
}
```

Рис. 3.15.Варіант динамічного задання конфігурації вікна.

Клас *MainWindowController* – прив'язаний до main-window.fxml за допомогою параметра fx:controller. Здійснює обробку даних введених користувачем в головне вікно, обробляє натискання клавіш та створює нові вікна передаючи туди дані введені користувачем.

Для обробки натискання клавіш створюються окремі методи які задаться у вищезгаданому fxml файлі ось так – *onAction=”#buttonClickMethod”.* Також обробник натискання на кнопку, або інший елемент який це підтримує, можна задати динамічно у коді програми(рис 3.16).

```
codeDistanceButton.setOnAction(actionEvent ->
        { code.sort();
          showSimpleInfoWindow("Code distance",
          String.valueOf(code.getDistance())); });
```

Рис. 3.16.Варіант динамічного задання функції обробника.

Клас *NewSearchController* – прив’язаний до файлу find-new-code.fxml. Створюється об’єктом *MainWindowController*. Відповідає за запуск алгоритму для пошуку нових кодів, відповідно до заданої користувачем розрядності та кодової відстані. Запускає алгоритму в новому потоці використовуючи стандартний клас FutureTask, який дозволяє зберігати повернені результати, та отримати їх пізніше у головному потоці. Для запуску нового потоку використовується об’єкт Thread. Обробляє помилки вводу данних.

Клас *LoadCodeController* – прив’язаний до файлу load-code.fxml. Створюється об’єктом *MainWindowController*. Відповідає за запуск алгоритму для продовження пошуку кодів, перед тим завантаживши збережені проміжні дані. Запуск як і в попередньому випадку, виконується в новому потоці. Обробляє помилки читання проміжних даних.

Клас *SearchCodeAbstractController* – є базовим класом для двох вищезгаданих класів, містить у собі загальну логіку та поля. Посилає сигнал зупинки алгоритму на вимогу користувача та обробляє отримані результати. Зупинка алгоритму може бути двох видів: зупинений користувачем та

завершився самостійно. Для того щоб відслідковувати дані ситуації створюється ще один потік з об’єктом інтерфейсу Runnable який зображено на рисунку 3.17. Цей потік створює обробник натискання клавіші тоді коли й запускає алгоритм пошуку коду.

```
protected Runnable printer = new Runnable() {
    @Override
    public void run() {
        while(!futureTask.isDone()) {
        }
        if (!resolver.wasInterrupted()) {
            Platform.runLater(() -> {
                computingLabel.setText("Computing finished.");
                try {
                    showCodeWindow(futureTask.get().iterator().next());
                } catch (Exception e) {
                    e.printStackTrace();
                    showErrorWindow("Unexpected error in algorithm");
                }
            });
        }
    }
};
```

Рис. 3.17.Об’єкт для відстежування роботи алгоритму.

Взаємодія процесів у програмі та пересилання сигналів зображено на рисунку 3.18. Під час запуску в головному потоці(Main Thread) алгоритму пошуку кодів, запускається два нові потоки: один з власне самим алгоритмом(Algorithm Thread) та інший у якому відстежується статус алгоритму(Check Thread). Завершиться алгоритм тоді, коли головний потік надішле сигнал зупинки(цей сценарій позначений штриховою лінією), або коли буде знайдено максимальний код. Після завершення алгоритму і його потік припинить своє виконання. У третьому потоці очікується саме завершення потоку алгоритма. Після цього перевіряється чи він був виконаний до кінця, тобто максимальний код був знайдений, якщо так то виводится повідомлення про це, та результат виконання алгоритму.

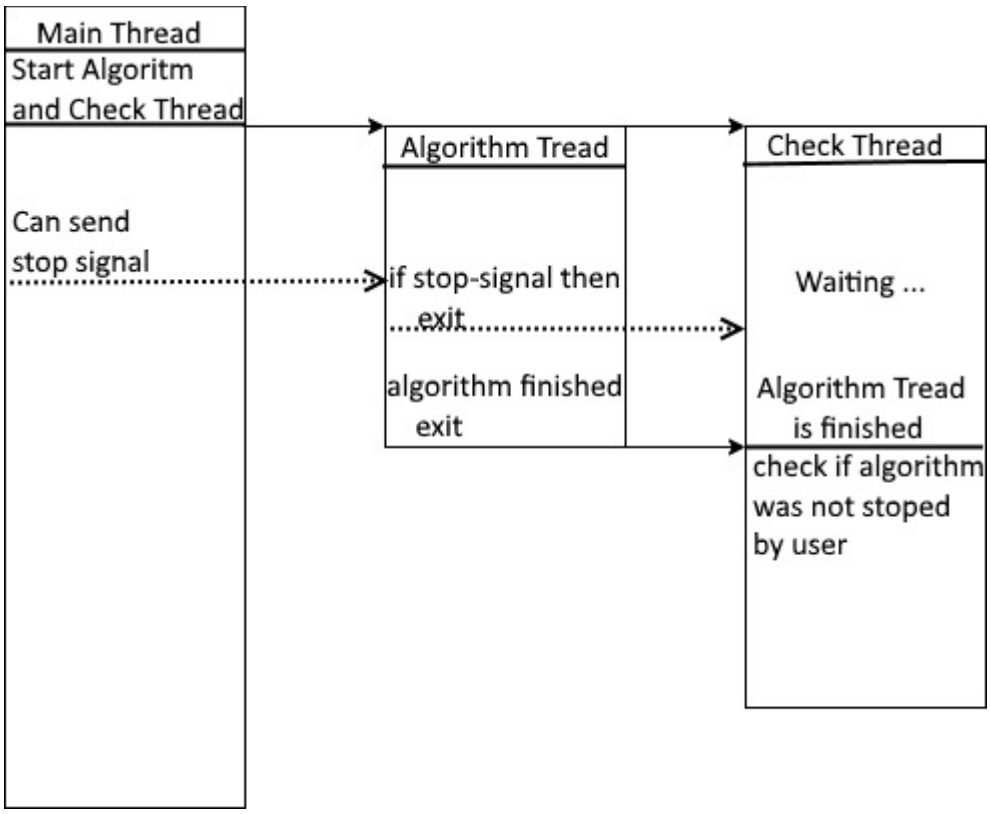


Рис. 3.18.Взаємодія потоків у програмі.

4. ТЕСТУВАННЯ КОМПЛЕКСУ ПРОГРАМ

4.3 Способи тестування програм.

Зараз існує безліч методів та інструментів тестування, які відрізняються залежно від потрібних цілей тестування(рис 4.1). Існує дві основні категорії – функціональне(functional) та не функціональне(not-functional) тестування. Функціональне тестування – полягає у тестуванні кожної окремої частини та функціональності великої системи, а не функціональне – це тестування всіх інших особливостей ПЗ, таких як швидкодія, безпека, зручність тощо. За способом виконання є ручне(*manual*) та автоматизоване(*automated*) тестування. Ручне тестування – це процес ручної перевірки программного забезпечення, всіх сценарії його роботи та знаходження помилок тестувальником, без використання додаткових засобів. Автоматизоване тестування – це процесс тестування, який частково або повністю автоматизований, за рахунок додаткових інструментів.

Модульне тестування (англ. unit testing) – полягає у тестуванні, перевірці коректності роботи окремого модуля, класу та методу або функції.

Інтеграційне тестування (англ. integration testing) – призначене для перевірки зв’язку між компонентами(модулями) системи, та коректності їх вза’ємодії між собою та з іншими системами.

Системне тестування (англ. system testing) – функціональне тестування системи в цілому, перевірка різних сценаріїв роботи системи в умовах наближених до реальних, у яких це ПЗ буде використовуватись.

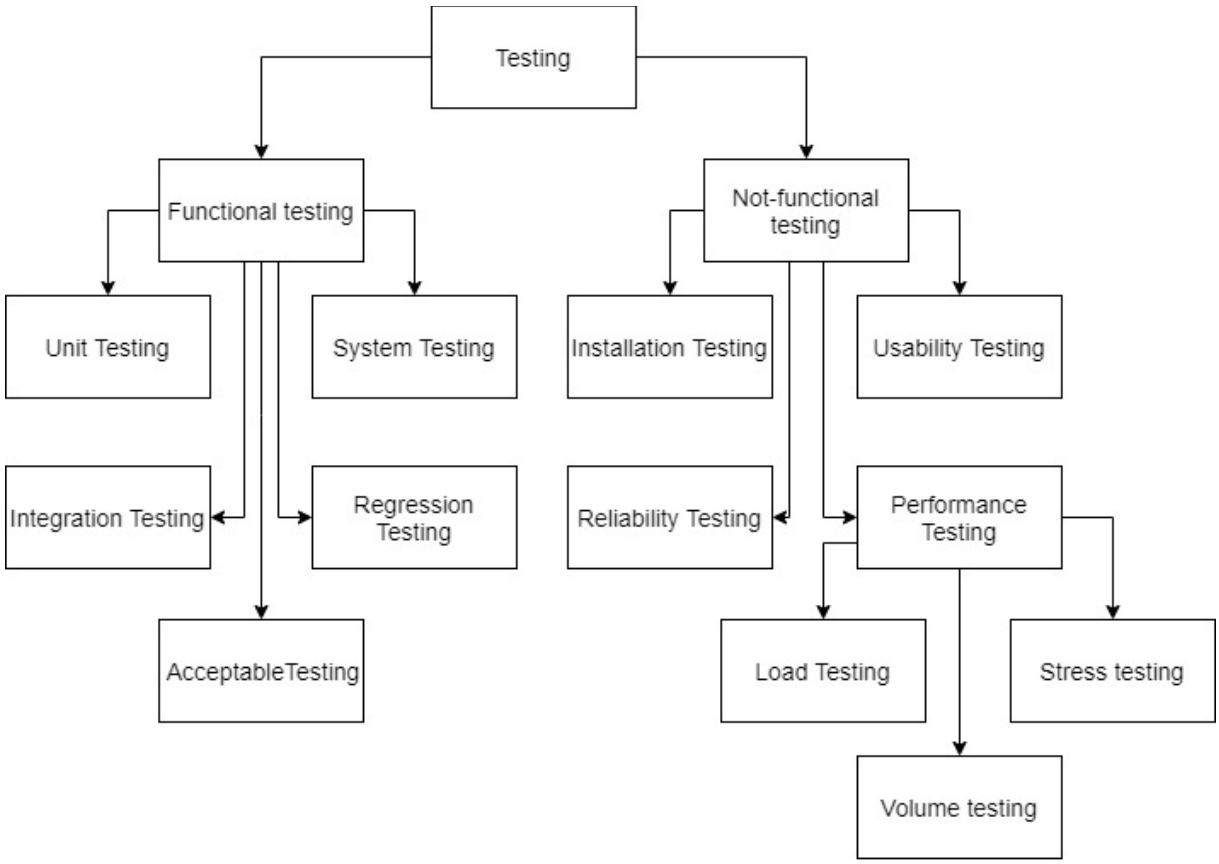


Рис. 4.1. Основні види тестування.

Регресивне тестування (англ. regression testing) – тестування програмного забезпечення після внесення яких-небудь змін.

Приймальне тестування (англ. acceptable testing) – перевірка системи на відповідність вимогам замовника.

Інсталяційне тестування (англ. installation testing) – тестування встановлення додатку та його налаштування. Багато сучасних програм встановлюються за допомогою спеціальних інсталяторів, які самі по собі також потребують тестування.

Тестування зручності використання (англ. usability testing) – це метод тестування, який встановлює степінь зручності, простоти та привабливості додатку для користувача в контексті заданих умов.

Тестування надійності (англ. reliability testing) – процес перевірки додатку з точку зору здатності успішно відновлюватися або забезпечувати коректну роботу під час виникнення різних можливих збоїв у роботі ПЗ або обладнання.

Навантажувальне тестування (англ. performance testing) – перевірка швидкодії системи при різних навантаженнях, визначення як система адаптується до різних навантажень, визначення задовільної границі цих навантажень.

4.4 Тестування модулів комплексу за допомогою JUnit.

В даній роботі було проведено модульне тестування, з використанням JUnit, який надає можливість простого створення тестів, їх запуску та перевірки результатів. JUnit – це інструмент для реалізації модульного тестування мовою програмування Java, який популяризує ідею керованої тестами розробки (англ. test-driven development) або TDD. Під час такого підходу спочатку створюються тести, які визначають функціональність та те як вона повинна працювати, а потім виконується розробка коду, який буде задовольняти відповідні тести.

Тест, або так званий тест-кейс, створюється шляхом додавання анотації @Test до методу. Тест-кейс - це частина коду, яка передбачає що інша частина коду, наприклад метод, працює як очікується. По суті, в тесті спочатку задаються різноманітні вхідні дані та очікуваний результат, після виконання певного коду отриманий результат порівнюється з очікуваним.

Тестування розробленого комплексу містить модульні тести які є частиною всього проекту(рис 4.2). Було створено тест кейси для тестування:

- пакету utils.
- пакет algorithm.

- пакету graph.

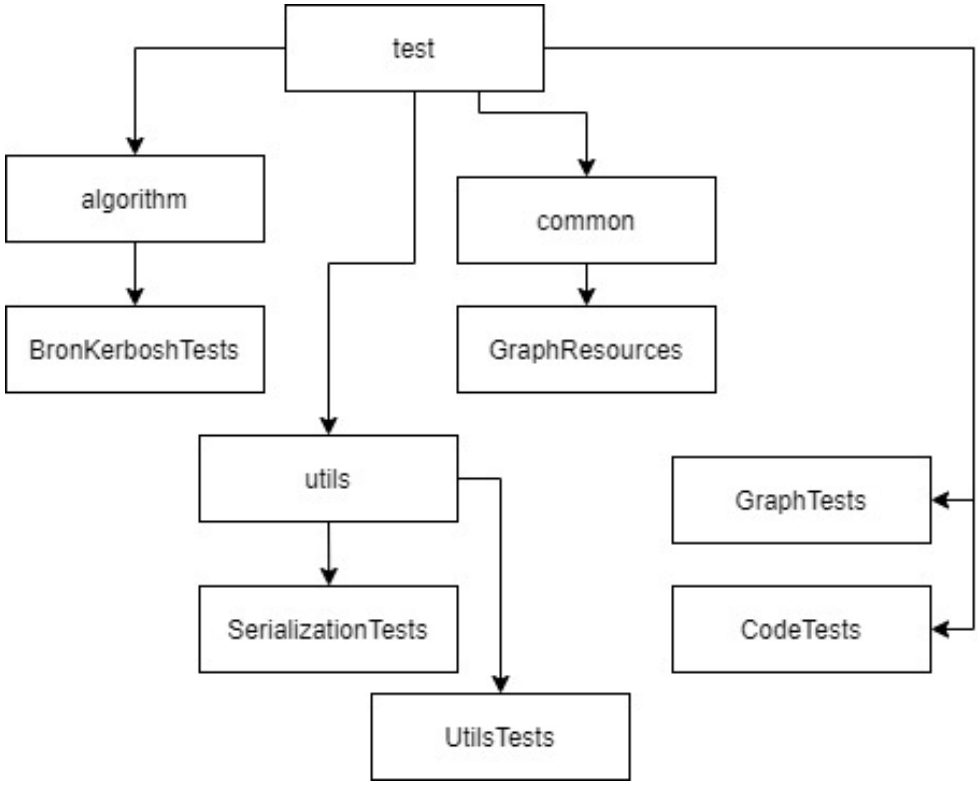


Рис. 4.2.Структура тестового модуля.

Пакет utils. Містить у собі два тестові класи.

SerializationTests – клас у якому реалізовано тест кейси для серіалізатора, перевірку правильності серіалізації та запису окремих об’єктів у файл. Подальше їх читання та тестування прочитаних об’єктів, на відповідність тим які були записані. Наприклад, метод тестування серіалізації об’єкта графу (рис 4.3). Також є тести для серіалізації множин та всього об’єкту проміжних даних алгоритму – IntermediateResult.

```java
@Test
public void serializationGraphTest() throws IOException {
    Graph g = new Graph();
    fillSimpleGraph(g);
    File tmp = File.createTempFile("testGraph", "tmp");
    tmp.deleteOnExit();
    Serializer.serializeGraph(g, tmp.getPath());
    Graph deserialize = Serializer.loadGraph(tmp.getPath());
    simpleGraphTest(deserialize);
}
```

Рис. 4.3.Метод тестування серіалізації графу.

UtilsTests – клас, що містить тест кейси для методів CodeUtils, а саме hemmingDistance та toBinary. Найпростіший приклад тесту з використанням JUnit на рисунку 4.4. Метод позначено анотацією @Test, тим самим вказючи для JUnit, що даний метод є тест-кейсом. Методи assertEquals, засіб для перевірки отриманого результату з відповідним очікуваним результатом. В даному тесті викликається функція hemmingDistance, яка власне і тестується, з різнимим вхідними параметрами, та перевіряється результат її повернення.

```java
@Test
public void hemmingDistanceTest() {
    Assertions.assertEquals(0, hemmingDistance(4, 4, 6));
    Assertions.assertEquals(2, hemmingDistance(11, 13, 4));
    Assertions.assertEquals(1, hemmingDistance(15, 31, 5));
}
```

Рис. 4.4.Метод тестування hemmingDistance.

Пакет algorithm містить клас BronKerboshTests із тестуванням алгоритму Брона-Кербоша.

У пакеті common знаходиться клас GraphResources, який містить дані та функції багаторазового використання, що використовуються в інших

тестових класах та запобігає дублюванню коду. В класі міститься загальний набір тестових вершин графа, та методи додавання вершин та ребер до графу, а також перевірки наявності конкретного ребра чи вершини у графі. Це використовується для тестування самого графа, а також під час тестування десеріалізованого графа.

Клас CodeTests релізує тестування функціональності класу Code:

- сортування коду.
- визначення мінімальної кодової відстані.
- визначення кодової відстані кодослова до коду.
- перетворення коду у двохвимірний масив.

Також тут використовується анотація @BeforeEach(рис 4.5), яка означає що, метод буде виконаний в перед кожним тест-кейсом. В цьому випадку для кожного тесту потрібний ініціалізований об'єкт Code, що забезпечує метод init у комбінації з вищезгаданою анотатцією. Існує і багато інших корисних анотацій, з якими можна ознайомитись в офіційній документації JUnit.

```
@BeforeEach
public void init() {
    List<Integer> codes = new ArrayList<>();
    codes.add(5);
    codes.add(0);
    codes.add(2);
    codes.add(8);
    code = new Code(6, codes);
}
```

Рис. 4.5.Метод ініціалізації коду.

Клас GraphTests тестує об’єкт типу Graph, простий граф, та алгоритм побудови графу Хемінга, з перевіркою присутності потрібних ребер між вершинами при заданій кодовій відстані, та відсутності непотрібних ребер. Також тестує перевизначені методи для компонентів графу Node та Edge. Наприклад методи equals та hashCode, які активно використовуються під час роботи з HashSet. І забезпечують правильне використання вищезгаданих компонентів графу, як елементів колекцій мови Java.

Запускати тести можна з багатьох сучасних IDE, для яких є спеціальні плагіни або ж із використанням збірників проектів, таких як gradle або maven.

Також під час розробки був використаний SonarLint, пагін для IDE, який по суті є статичним аналізатором коду, що може виявляти дефекти під час написання коду. Це дозволяє їх усунути до тестування, що значно спрощує процес розробки.

5. КЕРІВНИЦТВО КОРИСТУВАЧА

Розроблений комплекс програм для визначення нероздільних завадостійких кодів має назву Aqueduct. Написаний мовою програмування Java. Призначений для пошуку максимальних нероздільних завадостійких кодів з використанням достатньо ефективного алгоритму Брона-Кербоша. Який є удосконаленим для роботи з графом Хемінга, та працює практично в К(кількість вершин у графі) разів швидше.

Крім цього розроблений комплекс надає додаткову функціональність, таку як:

- Можливість задання конфігурації коду та запуск пошуку.
- Зупинка алгоритму в будь який момент, та збереження проміжних даних з якими він працював.
- Продовження роботи на основі збережених раніше даних.
- Визначення мінімальної відстань в коді.
- Відстань конкретного кодослова до коду.
- Сортування коду(перетворення його у впорядкований код)

Встановлення та використання комплексу.

Комплекс програм Aqueduct знаходиться у відкритому доступі із відкритим кодом, та може бути завантажений за посиланням: https://github.com/dedvolodya/Aqueduct.

Для роботи додатку необхідно мати встановлену JDK версії не старішої ніж 8. та збірник проектів gradle. Щоб запустити програму достатньо виконати команду: *./gradlew run*

Після запуску користувач може бачити головне вікно програми, зображене на рисунку 5.1.

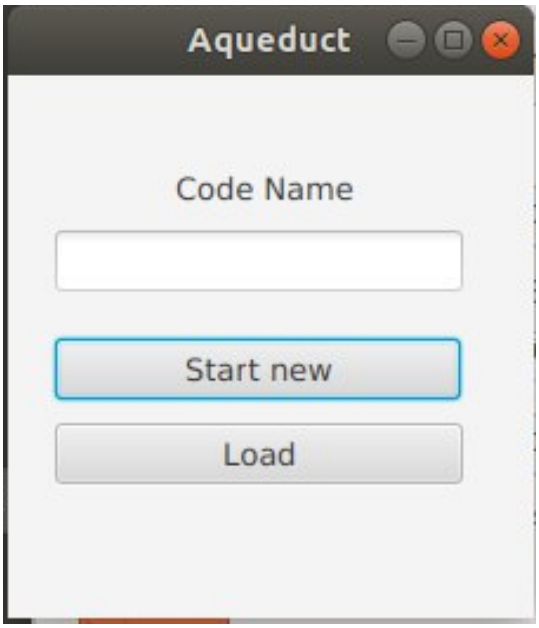


Рис. 5.1.Головне вікно програми.

В цьому вікні потрібно задати довільне ім’я коду. Оскільки програма може оперувати багатьма кодами, тобто можна запустити пошук, потім зберегти результати і почати пошук нового коду, а для того щоб повернутися до попередніх результатів використовується ім’я коду.

Вікно має дві кнопки: Start new – для запуску пошуку спочатку, Load – для завантаження збережених результатів.

Після натискання кнопки Start new відкривається наступне вікно(рис 5.2). У цьому вікні є два поля для введення даних: Bit Rate – для задання розрядності коду та Code distance – для задання мінімальної кодової відстані. Після введення даних у потрібно натиснути кнопку Start. Кнопка Stop в цьому режимі не активна. Після натискання кнопки старт відбудеться запуск алгоритму про що користувача буде сповіщено у цьому ж вікні(рис 5.3) повідомленням computing in progress….

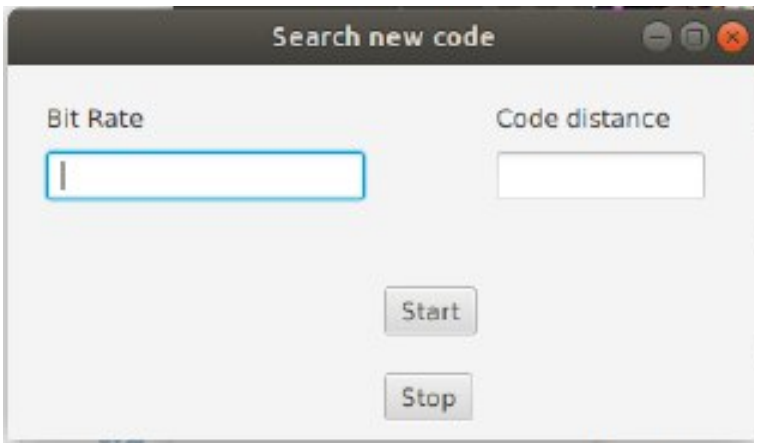


Рис. 5.2.Вікно пошуку нового коду.

Рис. 5.3.Виконання алгоритму.

Після цього маожна, чекати завершення алгоритму, для невеликих розрядностей пошук відбувається досить швидко, або зупинити алгоритм кнопкою Stop у разі потреби. В кожному випадку буде відображено відповідне повідомлення(рис 5.4, рис 5.5): *computing finished* – означає що алгоритм завершився успішно та *result saved* – означає що алгоритм був зупинений користувачем, а проміжні результати були збережені.

Рис. 5.4.Успішне завершення алгоритму.

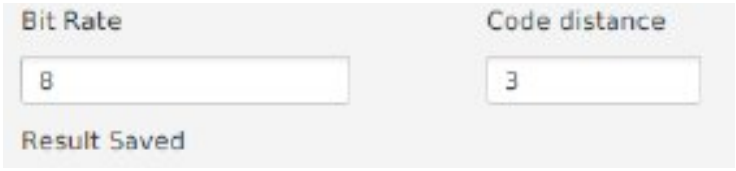


Рис. 5.5.Алгоритм був зупинений користувачем.

Відразу після цього у новому вікні буде виведено отримані результати(рис 5.6).

Рис. 5.6.Знайдений максимальний код.

Дане вікно містить матрицю, рядки якої є кодословами. Кожне кодослово позначене *word n*, де *n* - порядковий номер кодослова. Також в цьому вікні розміщені кнопки для додаткових операцій над кодом:

sort – перетворює код у впорядкований(рис 5.7).

code distance – обчислює мінімальну кодову відстань та відображає її у новому вікні(рис 5.8).

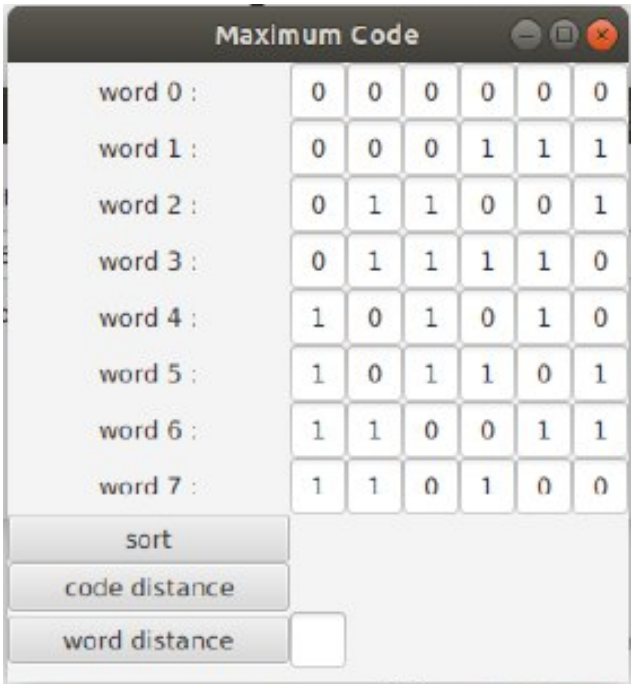


Рис. 5.7.Упорядкований максимальний код.

Рис. 5.8.Відображення кодової відстані.

word distance – обчислює мінімальну відстань Хемінга кодослова, індекс якого задається в комірці праворуч(рис 5.9) та відповідає певному кодословову із коду, до всього коду відображаючи результат в новому вікні(рис 5.10).

Рис. 5.9.Задання індексу кодослова.

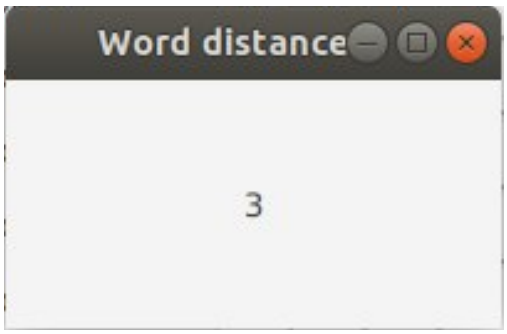


Рис. 5.10.Відображення відстані кодослова до коду.

Для завантаження збережених результатів та продовження роботи алгоритму, у головному вікні програми після введення імені коду необхідно натиснути кнопку Load. Після чого буде відкрите нове вікно(рис 5.11), подібне до вікна пошуку нового коду.

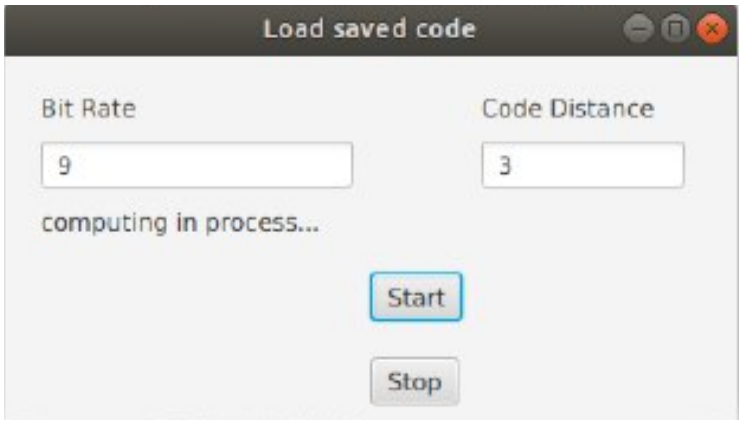


Рис. 5.11.Вікно завантаження результатів та продовження алгоритму.

Заголовок вікна в цьому випадку Load saved code, та значення в текстові поля задавати вручну не можливо, вони виводяться автоматично після натискання кнопки Start та показують характеристи коду, пошук якого було продовжено. Після запуску це вікно працює аналогічно, як уже вищезгадане. Для виходу з програми достатньо натиснути Stop, якщо алгоритм було запущено та позакривати вікна.

6. АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМИ

Було проведено серію запусків пошуку максимальних нероздільних завадостійких кодів з різними параметрами коду.

Розрядність – 6, кодова відстань – 3 (виправлення однієї помилки). Знайдений код на рисунку 6.1. Аналітична швидкість із формули (11) дорівнює 0.5

0	0	0	0	0	0
0	0	0	1	1	1
0	1	1	0	0	1
0	1	1	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	0	0

Рис. 6.1.Код C(8,6,3).

Аналогічним чином було знайдено коди з іншими розрядностями та відстанню Хемінга. Та було створено таблицю(рис 6.2), яка показує кількість кодослів та аналітичну швидкість знайдених комплексом кодів коду при заданій розрядності та кодовій відстані. Значення e – кратність помилки яку виправляє даний код, d – кодова відстань, n – розрядність коду, t – кількість кодослів знайденого коду.

Варто зауважити, що алгоритм для значень d=3 n=8 та d=3 n=9 не був завершений, результати були отримані після певного часу роботи алгоритму і подальшої зупинки.

e	d	n	t	(log t)/n
1	3	6	8	0.5
2	5	6	2	0.17
1	3	8	16	0.5
2	5	8	4	0.25
1	3	9	32	0.55
2	5	9	6	0.29

Рис. 6.2.Оцінка знайдених комплексом кодів з різними початковими значеннями.

Для порівняння ориманих результатів з уже існуючими нероздільними завадостійкими кодами для виправлення однієї помилки можна використати код Адамара C(8,7,4) , аналітична швидкість якого дорівнює 0.35 тоді, як код C(8,16,3) має швидкість 0.5. Проте в приведеному коді Адамара кодова відстань дорівнює 4, що дозволяє виявляти на одну помилку більше. Також існують коди з постійною вагою, які мають досить низьку надлишковість, наприклад аналітична швидкість коду C(7,35,2) дорівнює 0.73, проте такі коди мають ряд значних недоліків.

ВИСНОВОК.

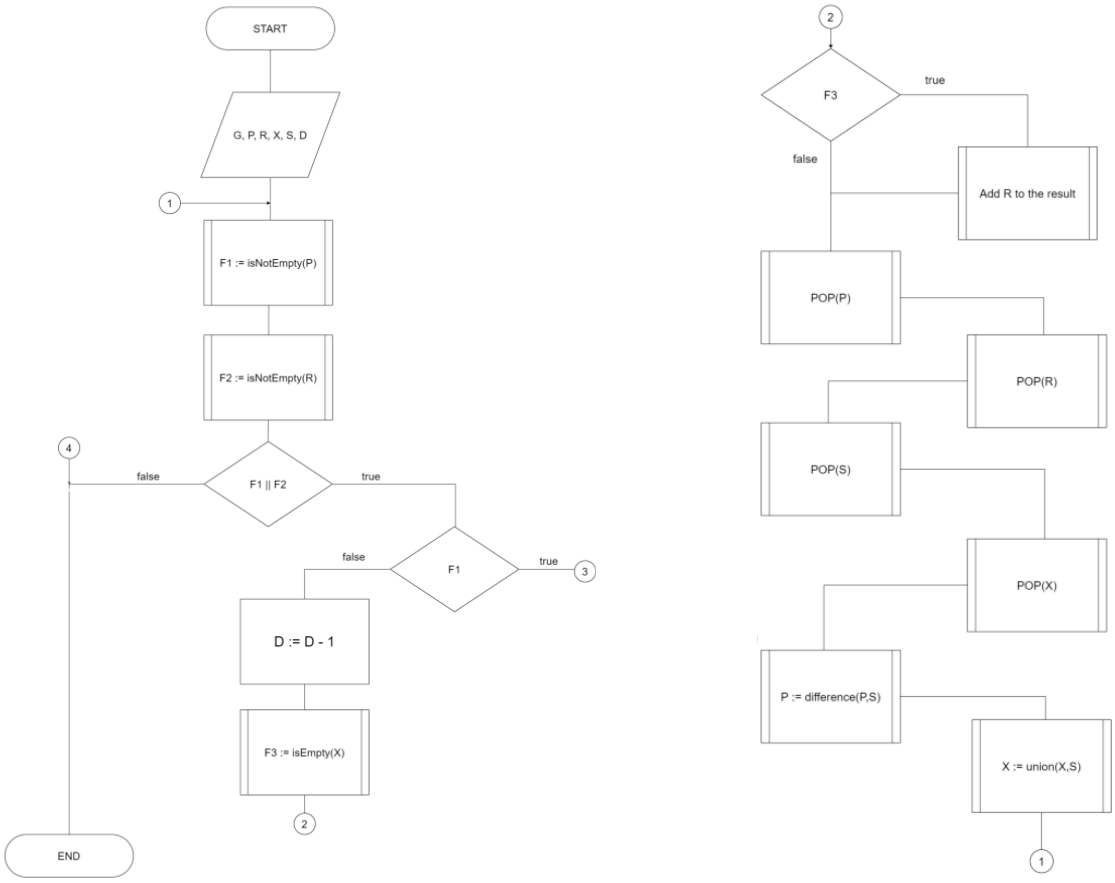
В даній роботі було розглянуто та проаналізовано проблеми пов’язані із завадостійким кодуванням, а саме нероздільними завадостікими кодами, переваги та недоліки алгоритму Брона-Кербоша. Також було розглянуто особливості графу Хемінга, та було запропоновано варіанти оптимізації алгоритму Брона-Кербоша враховуючи ці особливості та властивості еквіваленості кодів, що дозволило суттєво збільшити швидкодію алгоритму.

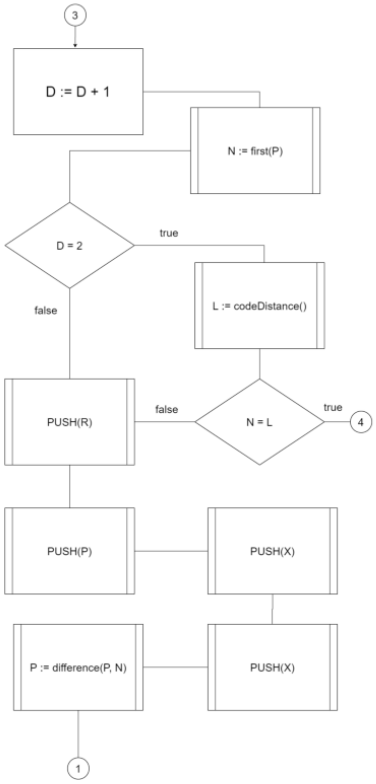
Було розроблено комплекс програм однією з найпопулярніших об’єктно орієнтовних мов програмування мовою Java. Перевагою такого вибору є те, що мова Java не є залежною від платформи і розроблені додатки легко переносяться та можуть запускатись на інших операційних системах.

Розроблений комплекс надає достатню функціональність для визначення максимальних нероздільних завадостійких кодів та виконання можливість виконання деяких простих операції над кодами. Розробка має простий та зрозумілий графічний інтерфейс і є простою в установці та використанні. Також, є можливість простого розширення функціональності та реалізації інших операцій над кодами.

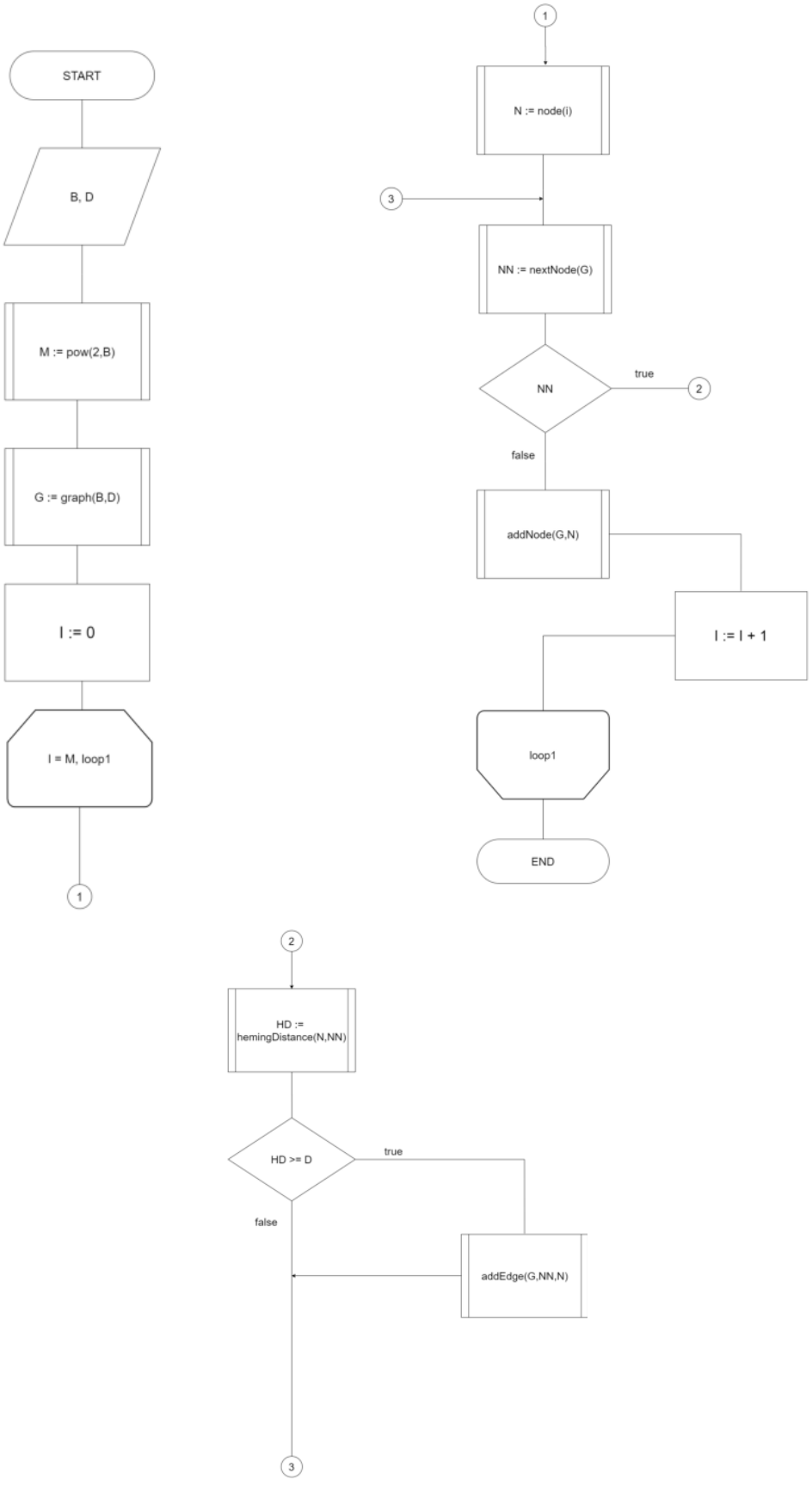
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ.

1. Березюк Н.Т. Кодирование информации 1978 – 201 с.
2. Ф. Дж. Мак-Вильямс, Н. Дж. А. Слоэн. Теория кодов, исправляющих ошибки 1979 – 52 с.
3. М. Н. Аршинов, Л. Е. Садовский. Коды и математика 1983 – 102 с.
4. Bron C. Finding all cliques of an undirected graph, 1973. – 575 с.
5. Free online encyclopedia – Wikipedia: Bron-Kerbosh Algorithm. URL : https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch_algorithm

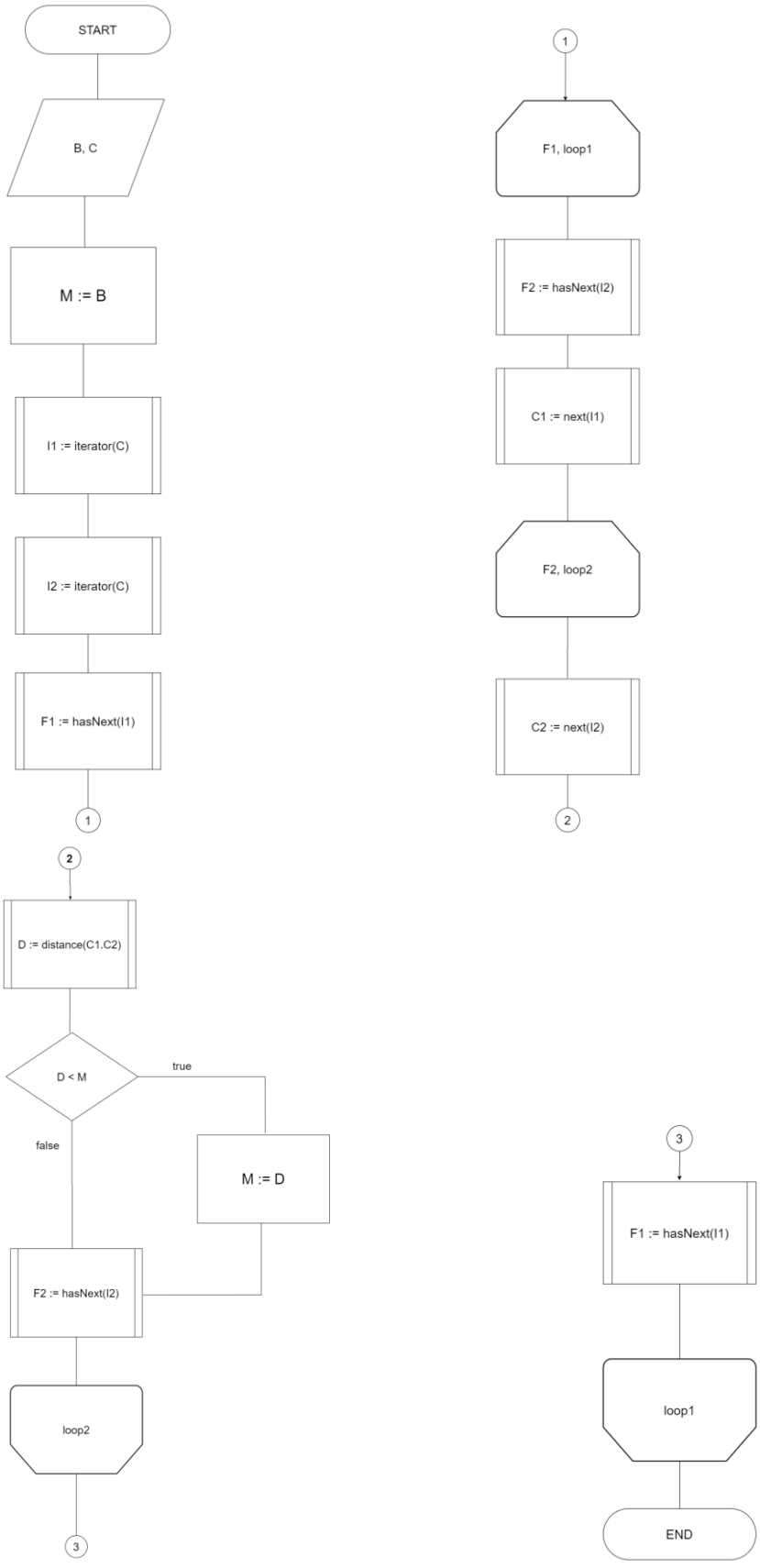




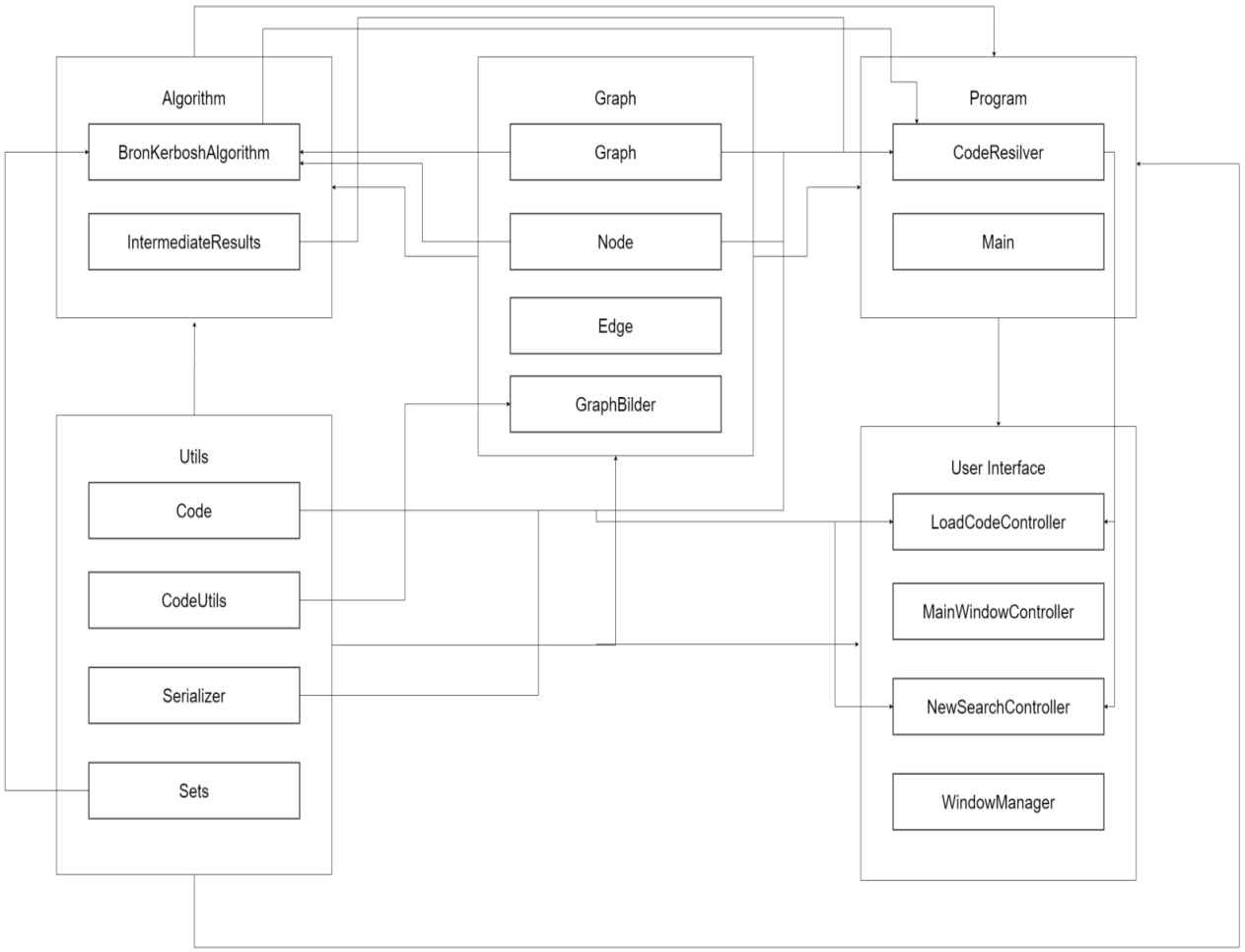
					ІАЛЦ.045490.005 Д1		
Зм.	**Арк.**	**№ докум.**	**Підп.**	**Дата**			
Розроб.		**Кравчук В.В.**			**Покращений алгоритм Брона-Кербоша**	**Літ.**	**Аркуш** 1 / **Аркушів** 1
Перевір.		**Тесленко О.К.**					
					Схема алгоритму	**КПІ ім. Ігоря Сікорського, ФПМ, КВ-63**	
Н.контр.		**Клятченко Я.М.**					
Затв.		**Романкевчи В.О.**					



					ІАЛЦ.045490.006 Д2		
Зм.	**Арк.**	**№ докум.**	**Підп.**	**Дата**			
Розроб.		**Кравчук В.В.**			**Алгоритм побудови графа Хемінга**	**Літ.**	**Аркуш** 1 / **Аркушів** 1
Перевір.		**Тесленко О.К.**					
					Схема алгоритму	**КПІ ім. Ігоря Сікорського, ФПМ, КВ-63**	
Н.контр.		**Клятченко Я.М.**					
Затв.		**Романкевич В.О.**					



					ІАЛЦ.045490.007 ДЗ		
Зм.	Арк.	№ докум.	Підп.	Дата			
Розроб.		Кравчук В.В.			Алгоритм визначення мінімальної кодової відстані	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Тесленко О.К.					
					Схема алгоритму	КПІ ім. Ігоря Сікорського, ФПМ, КВ-63	
Н.контр.		Клятченко Я.М.					
Затв.		Романкевич В.О.					



					ІАЛЦ.045490.008 Д4		
Зм.	**Арк.**	**№ докум.**	**Підп.**	**Дата**			
Розроб.		Кравчук В.В.			**Комплекс для визначення завадостійких кодів**	**Літ.**	**Аркуш** 1 / **Аркушів** 1
Перевір.		Тесленко О.К.					
					Схема структурна	**КПІ ім. Ігоря Сікорського, ФПМ, КВ-63**	
Н.контр.		Клятченко Я.М.					
Затв.		Романкевич В.О.					